Язык программирования C

ЛЕКЦИИ И УПРАЖНЕНИЯ

6-е издание

C Primer Plus

Sixth Edition

Stephen Prata

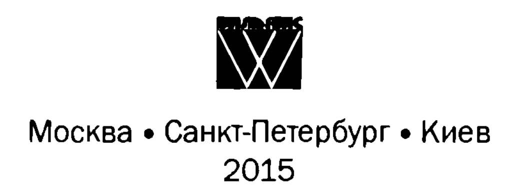


Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Язык программирования С
ЛЕКЦИИ И УПРАЖНЕНИЯ

6-е издание

Стивен Прата



Москва • Санкт-Петербург • Киев
2015

ББК 32.973.26-018.2.75
П70
УДК 681.3.07

Издательский дом Вильямс
Зав. редакцией *С.Н. Тригуб*
Перевод с английского *Ю.Н. Артеменко*
Под редакцией *Ю.Н. Артеменко*

По общим вопросам обращайтесь в Издательский дом “Вильямс” по адресу:
info@williainspublishing.coin, http://www.williamspublishing.com

Прата, Стивен.

П70 Язык программирования С. Лекции и упражнения, 6-е изд. : Пер. с англ. —М. : ООО “И.Д. Вильямс”, 2015. — 928 с. : ил. - Парал, тит. англ.

ISBN 978-5-8459-1950-2 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Стивен Прата

Язык программирования С. Лекции и упражнения
6-е издание

Верстка *Т.Н. Артеменко*
Художественный редактор *В.Г. Пашютин*

Подписано в печать 24.02.2015. Формат 70x100/16.
Гарнитура Times.
Уел. печ. л. 74,82. Уч.-изд. л. 54,2.
Тираж 500 экз. Заказ № 867.

Отпечатано способом ролевой струйной печати
в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.iu. E-mail: sales@chpd.nj, тел.: 8(499)270 73 59

ООО “И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1950-2 (рус.)
ISBN 978-0-321-92842-9 (англ.)

Оглавление

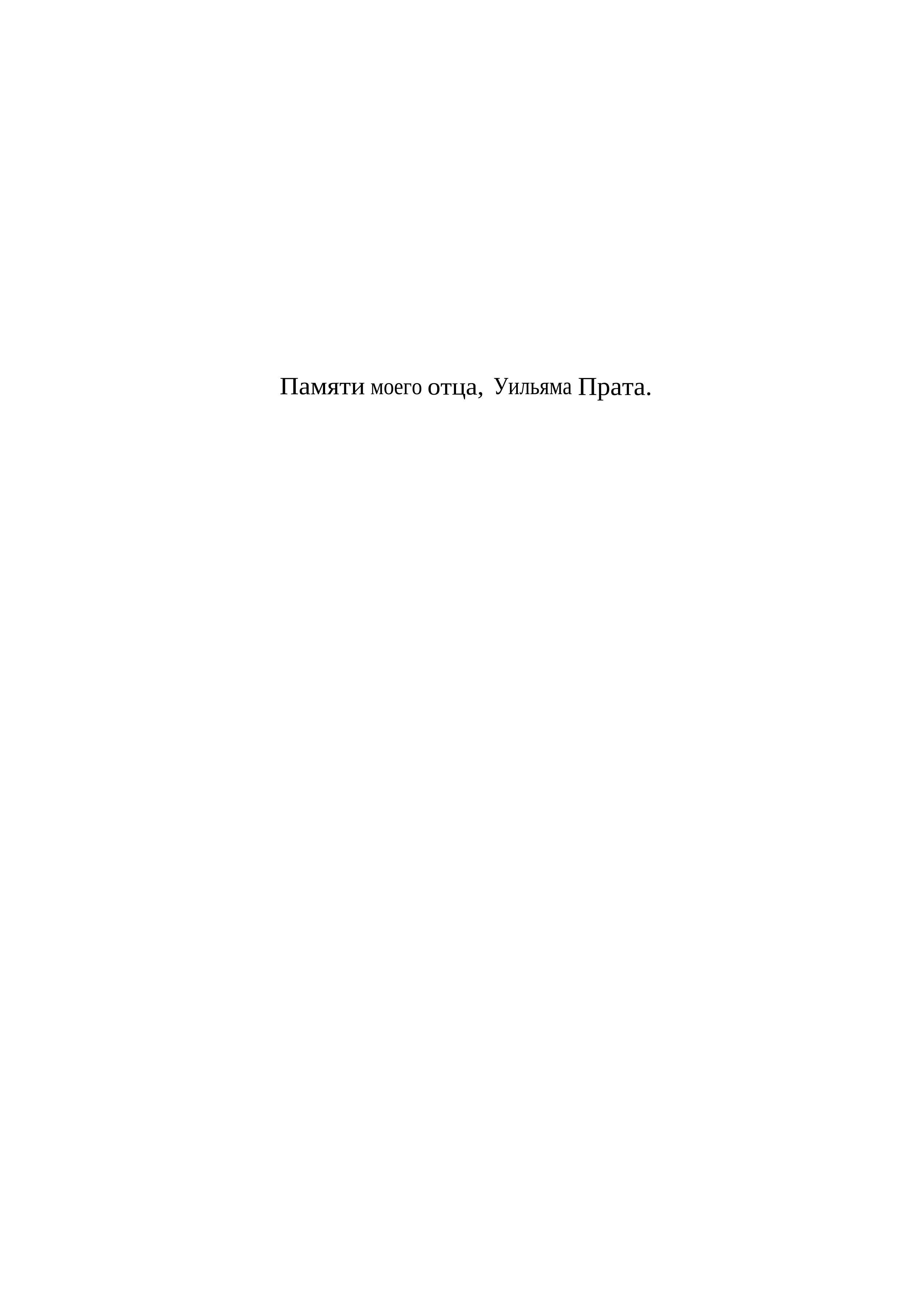
Содержание

Глава 4. Символьные строки и форматированный ввод-вывод 117

Глава 5. Операции, выражения и операторы 157

Глава 6. Управляющие операторы C: циклы 199

Глава 7. Управляющие операторы C: ветвление и переходы 247


Памяти моего отца, Уильяма Прата.

Об авторе

Стивен Прата, в настоящее время отошедший от дел, преподавал астрономию, физику и программирование в Колледже Марин в Кентфилде, штат Калифорния. Он получил диплом бакалавра в Калифорнийском технологическом институте и степень доктора философии в Калифорнийском университете в Беркли. Его увлечение компьютерами началось с компьютерного моделирования звездных скоплений. Стивен является автором и соавтором более десятка книг, включая C++ *Primer Plus (Язык прогр^амирования* C++. *Лекции и упражнения, 6-е изд*, ИД "Вильяме", 2012 г.) и *Unix Primer Plus.*

Благодарности

Я хотел бы поблагодарить Марка Табера за продолжение этого проекта и доведение его до конца. Также я благодарен Денни Калев за техническую помощь и за предложенный им термин “область действия программы”.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно бьию сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш всб-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

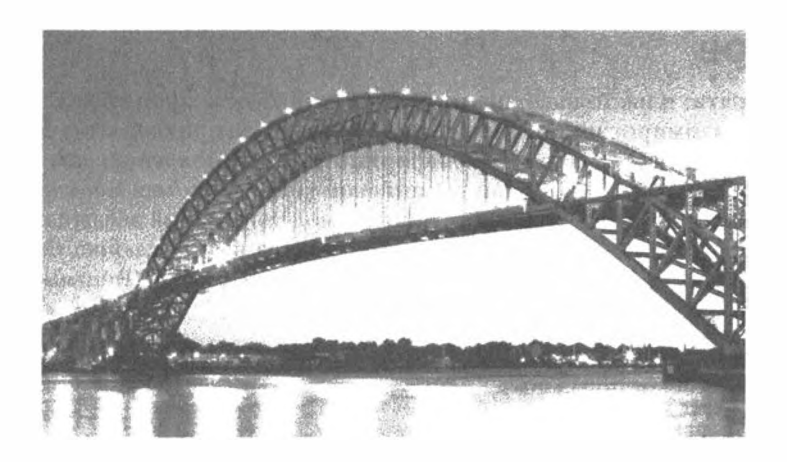
Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:

E-mail: info@williamspublishing.com
http://www.williamspublishing.com

Информация для писем из:

России: 127055, i: Москва, ул. Лесная, д. 43, стр. 1
Украины: 03150, Киев, а/я 152



Бейоннский мост, соединяющий Бейонн, штат Нью-Джерси, со Статен-Айлендом, Нью-Йорк, был самым длинным в мире стальным арочным мостом, когда его открыли в 1931 году, и удерживал эту позицию на протяжении 45 лет. В наши дни многие по-прежнему считают его значительным эстетическим и техническим достижением.

С пролетом в 511 метров грациозная арка моста вздымается на высоту 69 метров над проливом Килл-Ван-Кул и позволяет беспрепятственно проходить судам по бухте Ньюарк, главном судоходном канале к островным портам Ньюарка и Элизабета, штат Нью-Джерси.

Инженерные достижения Бейоннского моста включают использование впервые в мире марганцевой стали для основных структурных элементов, значительный прогресс в структурном анализе и инновационную систему строительных лесов, примененных при его возведении.

Проектировщик моста, Отмар Амманн, выбрал элегантный дизайн со стальными арками, отбросив вариант с консольно-подвесными строениями как слишком дорогостоящий и непрактичный для той местности.

В 1931 году Американский институт стальных конструкций присудил этому мосту приз как самому красивому стальному арочному мосту, а в "Нью-Йорк таймс", запоздало отдавая дань, отметили, что Бейоннскому мосту присуща симметрия и плавность деталей, которые производят глубокое впечатление и надолго западают в память.

В 2013 году портовые власти Нью-Йорка и Нью-Джерси запустили проект реконструкции моста на сумму 1,3 миллиарда долларов для увеличения высоты пролета в пределах существующей структуры арки, чтобы крупные контейнеровозы могли проходить под мостом транзитом в порты Ньюарка и Элизабета.

Предисловие

Когда в 1984 году было вышло первое издание этой книги, С был относительно малоизвестным языком программирования. С тех пор начался бурный рост языка, и многие изучали С с помощью этой книги. В действительности было продано свыше 550 000 экземпляров книги в ее различных изданиях.

С развитием языка от раннего неформального стандарта K&R, через стандарт 1990 года ISO/ANSI и стандарт 1999 года ISO/ANSI, до появления в 2011 году стандарта ISO/IEC, обретала зрелость и данная книга, добравшись до своего шестого издания. Как и во всех предшествующих изданиях, моей целью было создание поучительного, ясного и полезного введения в язык С.

Подход и цели

Эта книга должна служить дружественным, простым в использовании и пригодным для самостоятельного обучения руководством. Чтобы соответствовать этой цели, в книге задействована описанная ниже стратегия.

- Наряду с описанием возможностей языка С объясняются концепции программирования; в книге *е* предполагается, что вы являетесь профессиональным программистом.
- Множество приведенных в книге коротких примеров, которые легко вводить с клавиатуры, иллюстрируют одну или две концепции за раз, поскольку обучение через практику представляет собой один из наиболее эффективных способов усвоения новой информации.
- Рисунки и иллюстрации проясняют концепции, которые трудно описать только одними словами.
- Главные средства языка С подытожены во врезках, на которые легко ссылаться и пересматривать.
- Вопросы для самоконтроля и упражнения по программированию в конце каждой главы позволяют проверять и закреплять понимание языка С.

Чтобы извлечь максимальную пользу, при изучении материала данной книги вы должны играть как можно более активную роль. Не ограничивайтесь одним лишь чтением примеров — вводите и опробуйте их в своей системе. Язык С является в высокой степени переносимым, но вы можете обнаружить отличия между тем, каким образом программа ведет себя в вашей системе и как она работает у нас. Смело экспериментируйте, изменяя разные части программы, чтобы посмотреть, к чему это приведет. Модифицируйте программу, чтобы она делала что-то другое. Ищите возможность разработки альтернативного подхода. Игнорируйте появляющиеся временами предупреждения и исследуйте, что произойдет, если вы будете поступать некорректно. Попытайтесь ответить на вопросы и выполнить упражнения. Чем больше вы сделаете самостоятельно, тем большему вы научитесь и больше запомните.

Я надеюсь, что вы найдете это новое издание книги интересным и эффективным введением в язык программирования С.

1

Предварительные сведения

В ЭТОЙ ГЛАВЕ...

- Возможности и история создания языка С
- Действия, которые нужно выполнить для написания программ
- Немного о компиляторах и компоновщиках
- Стандарты языка С

Добро пожаловать в мир С — мощного языка программирования для профессионалов, который в равной степени популярен как в среде любителей, так и в среде программистов, пишущих программы для коммерческого применения. Данная глава подготовит вас к изучению и использованию этого мощного и широко распространенного языка и ознакомит с различными операционными средами, в которых, скорее всего, вам придется совершенствовать свои знания языка С.

Прежде всего, мы ознакомимся с происхождением языка С и исследуем некоторые его свойства, а также его сильные и слабые стороны. Затем мы изучим основы программирования и рассмотрим некоторые общие принципы программирования. В завершение мы обсудим, как выполнять программы на языке С в ряде известных систем.

Появление языка С

Деннис Ритчи из компании Bell Labs создал язык С в 1972 году, когда они вместе с Кеном Томпсоном работали над созданием операционной системы Unix. Однако сам язык С зародился в голове Ритчи не просто так. Его предшественником был язык B, созданный Томпсоном на основе..., впрочем, это уже другая история. Наиболее важным является тот факт, что С задумывался как инструментальное средство для программистов-практиков, следовательно, его главной целью в этом случае было создание полезного языка программирования.

Большинство языков программирования создавались с целью быть полезными, но довольно часто перед ними ставились другие вопросы. Например, изначально язык Pascal предназначался для облегчения изучения принципов программирования. С другой стороны, язык BASIC создавался как язык программирования, приближенный к естественному английскому языку, чтобы облегчить задачу изучения языков программирования для студентов, не знакомых с компьютерами. Это были важные цели, но они не всегда соответствовали прагматичном' подходу к решению повседневных задач. Тем не менее, разработка С как языка, предназначенного для программистов, сделала его одним из наиболее востребованных в настоящее время.

Причины популярности языка С

В течение последних четырех десятилетий С стал одним из основных и наиболее широко распространенных языков программирования. Его популярность росла потому, что люди предпринимали попытки работать с ним и убеждались в его достоинствах. За последнее десятилетия или два многие программисты перешли на такие языки, как C++, Objective С и Java, но язык С вес еще остается важным и сам по себе, и как путь перехода на указанные языки. По мере изучения С вы убедитесь, что он обладает многими достоинствами (рис 1.1). Некоторые из них мы отметим сейчас.

Конструктивные особенности

С представляет собой современный язык программирования, включающий управляющие средства, которые теория и практика вычислительной техники рассматривает как полезные и желательные. Его конструкция хорошо подходит для нисходящего планирования, для структурного программирования и для модульного проектирования. Все это позволяет получать надежные и понятные программы.



Мощные управляющие структуры



Высокая скорость



Компактный код — небольшие по объему программы



Переносимость на другие компьютеры

Рис. 1.1. Достоинства языка C

Эффективность

C является эффективным языком программирования. Его конструкция продуктивно использует возможности компьютеров, на которых он установлен. Программы на C отличаются компактностью и быстротой исполнения. По сути дела C обладает некоторыми средствами точного управления, обычно характерными разве что для языка ассемблера. (*Язык ассемблера* — это мнемоническое представление множества инструкций, используемых конкретным центральным процессором; различные семейства центральных процессоров имеют разные языки ассемблера.) При желании программы можно настроить на максимальную скорость выполнения или на более эффективное использование памяти.

Переносимость

C является переносимым языком, и это означает, что программу, написанную на C для одной системы, можно выполнять на другой системе всего лишь с небольшими изменениями, а иногда удается обойтись вообще без модификаций. В тех случаях, когда изменения неизбежны, они ограничиваются простым редактированием нескольких записей в заголовочном файле, сопровождающем главную программу. Многие языки декларируются как переносимые, однако тем, кто преобразовывал программу на языке BASIC, предназначенном для ПК компании IBM в программу на языке BASIC для компьютера Apple (они были близкими родственниками), либо предпринимал попытки выполнить в среде Unix программу на языке FORTRAN, которая предназначена для

мэйнфрейма IBM, хорошо известно, что такой перенос – в лучшем случае весьма трудоемкая операция. Язык C является лидером в смысле переносимости. Компиляторы языка C (программы, преобразующие код на C в инструкции, которые компьютер использует для внутренних целей) доступны для многих компьютерных архитектур, от 8-разрядных микропроцессоров до суперкомпьютеров Cray. Однако следует отметить, что фрагменты программы, написанной специально для доступа к конкретным аппаратным устройствам, таким как монитор или специальные функции операционных систем, подобных Windows 8 или OS X, обычно не принадлежат к числу переносимых.

Поскольку язык C тесно связан с Unix, операционные системы семейства Unix поставляются с компилятором C в виде части соответствующего пакета. Установка операционной системы Linux также обычно включает компилятор языка C. Доступно несколько компиляторов языка C, предназначенных для персональных компьютеров, в том числе для работающих под управлением различных версий ОС Windows и Macintosh. Таким образом, используете вы домашний компьютер, профессиональную рабочую станцию или мэйнфрейм, у вас высокие шансы получить компилятор языка C для вашей конкретной системы.

Мощь и гибкость

Язык C является мощным и гибким (это два наиболее предпочитаемых определения в литературе компьютерной тематики). Например, большая часть кода мощной и гибкой операционной системы Unix была написана на C. На языке C были реализованы многие компиляторы и интерпретаторы для других языков, таких как FORTRAN, Perl, Python, Pascal, LISP, Logo и BASIC. В результате, когда вы используете FORTRAN на машине Unix, в конечном итоге именно программа, написанная на C, выполняет работу по созданию окончательной исполняемой программы. Программы на C применялись для решения физических и инженерных задач и даже для анимации специальных эффектов для множества фильмов.

Ориентация на программистов

Язык C ориентирован на удовлетворение потребностей программистов. Он предоставляет вам доступ к оборудованию и позволяет манипулировать отдельными фрагментами памяти. Он также предоставляет богатый выбор операций, которые позволяют лаконично выражать свой подход к решению задач. В плане ограничения того, что можно делать, язык C менее строг, чем, скажем, Pascal или даже C++. Такая гибкость является достоинством и одновременно представляет определенную опасность. Достоинство заключается в том, что решать многие задачи, такие как преобразование форматов данных, в C намного проще, чем в других языках. Опасность состоит в том, что есть шанс допускать такие ошибки, которые в других языках попросту невозможны. Язык C предоставляет большую свободу действий, но при этом налагает и более высокую ответственность.

Наряду с этим, большинство реализаций языка C сопровождаются обширной библиотекой полезных функций на C. Эти функции способны удовлетворить многие потребности, с которыми сталкивается программист.

Недостатки

Язык C не лишен недостатков. Часто, как это бывает у людей, недостатки и достоинства являются противоположными сторонами одного и того же свойства. Например, как мы уже упоминали, свобода выражений в языке C также требует дополнительной ответственности. В частности, использование в C указателей (одна из

многочисленных тем, которые будут рассматриваться в настоящей книге) означает возможность появления программных ошибок, которые трудно отследить. Как отметил один из известных людей, ценой свободы является постоянная бдительность.

Выразительность языка С в сочетании с богатством его операций делает возможным написание кода, который исключительно сложно понять. Конечно, вы отнюдь не обязаны писать неясный код, но такая возможность имеется. В конце концов, для какого еще языка устраивается ежегодный конкурс на самый запутанный код?

В языке С много достоинств, но, несомненно, не меньше и недостатков. Однако вместо того, чтобы углубляться в эти материи, давайте перейдем к новой теме.

Происхождение языка С

В начале восьмидесятых годов прошлого столетия С уже был доминирующим языком программирования в среде миникомпьютеров, функционировавших под управлением операционных систем Unix. С тех пор он распространился на персональные компьютеры (микрокомпьютеры) и мэйнфреймы (большие вычислительные машины). Взгляните на рис. 1.2. Многие компании по разработке и поставке программного обеспечения предпочитают использовать именно язык С при создании программ для текстовых процессоров, крупномасштабных электронных таблиц, компиляторов и других программных продуктов. Эти компании убедились в том, что с помощью С можно создавать компактные и эффективные программы. А еще важнее то, что эти в программы легко вносить изменения и легко адаптировать к новым моделям компьютеров.



Рис. 1.2. Где используется язык С

Все, что хорошо для компаний и ветеранов языка С, хорошо также и для других пользователей. Все больше и больше пользователей компьютеров обращаются к языку С, чтобы задействовать его преимущества. Для программирования на языке С вовсе не надо быть компьютерным профессионалом.

В девяностых годах прошлого столетия многие компании, изготавливающие и поставляющие программное обеспечение, при реализации крупных программных проектов стали переходить на язык С++. Язык С++ добавляет к С инструментальные средства объектно-ориентированного программирования. (Объектно-ориентированное программирование представляет собой философию, которая пытается формировать язык таким образом, чтобы он соответствовал задаче, в отличие от формулирования задачи так, чтобы она соответствовала языку программирования.) В первом приближении С++ можно рассматривать как надмножество языка С в том смысле, что программа на С также является или почти является программой на С++. Изучая язык С, вы фактически изучаете многие аспекты С++.

Несмотря на популярность более новых языков вроде С++ и Java, язык С сохраняет лидирующее положение по способности решать задачи из области разработки программного обеспечения, обычно входя в десятку наиболее востребованных языков программирования. В частности, С неизменно используется для программирования встроенных систем. Иначе говоря, он все чаще применяется для программирования обычных микропроцессоров, встроенных в автомобили, камеры, DVD-проигрыватели и другие современные бытовые устройства. Наряду с этим С посягает на долговременное господство языка FORTRAN в области научного программирования. И, наконец, как язык, создававшийся для разработки операционных систем, он играет ключевую роль в построении операционной системы Linux. Таким образом, и во второй декаде двадцать первого века С продолжает удерживать за собой сильные позиции. Короче говоря, С является одним из наиболее важных языков программирования и надолго останется таковым. Если вы хотите заниматься разработкой программ, то на вопрос, можете ли вы работать на языке С, вы непременно должны ответить утвердительно.

Особенности функционирования компьютеров

Прежде чем приступать к изучению программирования в языке С, вероятно, следует иметь хотя бы самое общее представление о том, как работает компьютер. Эти знания помогут понять, какова связь между написанием программы на С и тем, что на самом деле происходит при ее выполнении.

Современные компьютеры состоят из нескольких компонентов. *Центральный процессор* (ЦП) выполняет основную вычислительную работу. *Память с произвольным доступом*, или *оперативное запоминающее устройство* (ОЗУ), представляет собой рабочую область, в которой содержатся программы и файлы. Постоянное запоминающее устройство (в прошлом им, как правило, был жесткий диск, но теперь все чаще и чаще его роль исполняет твердотельный диск) хранит все эти программы и файлы, даже когда компьютер выключен. Периферийные устройства различного назначения, такие как клавиатура, мышь, сенсорный экран и монитор, обеспечивают обмен данными между пользователем и компьютером. ЦП обрабатывает программы, поэтому рассмотрим его роль подробнее.

Функции ЦП, по крайней мере, в таком упрощенном представлении архитектуры компьютера, достаточно просты. Процессор извлекает команду из памяти и выполняет ее. Затем он извлекает следующую команду и выполняет ее, и т.д. (ЦП с тактовой частотой 1 ГГц выполняет порядка одного миллиарда таких операций в секунду, так что ЦП ведет монотонную жизнь, но в бешеном темпе.) ЦП имеет собственную

рабочую область, состоящую из нескольких *регистров*, каждый из которых может запоминать какое-то число. Один регистр содержит адрес памяти следующей команды, и ЦП использует эту информацию для извлечения следующей команды. После извлечения следующей команды ЦП запоминает ее в другом регистре и обновляет первый регистр адресом очередной команды. Центральный процессор выполняет ограниченный набор команд (получивший название *набора инструкций*). Наряду с этим, его команды достаточно специфичны; многие из них требуют от ЦП перемещения числа из одного места в другое — например, из ячейки памяти в регистр.

Здесь следует отметить два интересных обстоятельства. Во-первых, все, что хранится в компьютере, хранится в виде чисел. Числа сохраняются как числа. Символы, такие как буквы алфавита, используемые в текстовых документах, сохраняются как числа, при этом каждый символ обладает своим числовым кодом. Команды, которые компьютер загружает в свои регистры, сохраняются как числа, и каждая команда из набора инструкций имеет числовой код. Во-вторых, компьютерная программа в конечном итоге должна быть выражена в этом числовом коде, или, другими словами, с помощью *машинного языка*.

Одним из последствий такого принципа работы компьютера является то, что если вы хотите, чтобы компьютер выполнил какую-то работу, то должны ввести конкретный список инструкций (программу), подробно расписывающий, что и как нужно сделать. Вы должны создать программу на языке, который понятен непосредственно компьютеру (на машинном языке). Это кропотливая и утомительная работа, требующая большой точности. Простая операция вроде сложения двух чисел должна быть разбита на несколько шагов, примерно так, как описано ниже.

1. Скопировать число из ячейки памяти 2000 в регистр 1.
2. Скопировать число из ячейки памяти 2004 в регистр 2.
3. Сложить содержимое регистра 2 с содержимым регистра 1 и оставить результат сложения в регистре 1.
4. Скопировать содержимое регистра 1 в ячейку памяти 2008.

И каждую из этих инструкций придется представить в числовом коде!

Если написание программ в таком стиле вам нравится, то вы огорчитесь, узнав, что золотой век программирования в машинных кодах давно канул в прошлое. Однако если вы предпочитаете что-то более интересное, откройте свое сердце языкам программирования высокого уровня.

Языки программирования высокого уровня и компиляторы

Языки программирования высокого уровня, такие как С, существенно упрощают вашу жизнь как программиста несколькими способами. Во-первых, вы не должны представлять команды в числовом коде. Во-вторых, команды, которые вы используете, намного ближе к тому, каким образом вы думаете о задаче, нежели к тому, как она представлена в рамках детализированного подхода, применяемого компьютером. Вместо того чтобы обременять себя мыслями о том, какие действия конкретный ЦП должен предпринять, чтобы решить конкретную задачу, вы можете выразить свои пожелания на более абстрактном уровне. Чтобы сложить два числа, вы можете, например, написать код следующего вида:

```
total = mine + yours;
```

Видя код, подобный этому, вы сразу же догадываетесь, что он делает, в то же время, просматривая эквивалентный код на машинном языке, который содержит несколько команд, выраженных в числовой форме, трудно сходу понять, о чем идет речь.

К сожалению, для компьютера все происходит с точностью до наоборот: для него команда на языке высокого уровня — непонятная бессмыслица. Именно в этот момент в игру выступают компиляторы. *Компилятор* — это программа, которая переводит программу, представленную на языке высокого уровня, в детальный набор команд на машинном языке, понимаемых компьютером. Вы формулируете задачу на высоком уровне, а компилятор берет на себя заботу об остальных скучных деталях.

Подход с использованием компилятора дает еще одно преимущество. В общем случае каждый компьютер обладает собственным уникальным машинным языком. Поэтому программа, написанная на машинном языке, например, для ЦП Intel Core i7, совершенно бессмысленна для процессора с ARM-архитектурой Cortex-A57. В то же время компилятор можно приспособить для конкретного машинного языка. Следовательно, располагая нужным компилятором или набором компиляторов, можно преобразовывать одну и ту же программу на языке высокого уровня в разнообразные программы на разных машинных языках. Вы решаете задачу программирования только один раз, после чего предоставляете компиляторам возможность транслировать ее решение на множество различных машинных языков.

Короче говоря, языки высокого уровня, такие как С, Java и Pascal, описывают действия в более абстрактной форме и не привязаны к конкретному ЦП или набору инструкций. Кроме того, языки высокого уровня проще изучать, и на них намного легче писать программы, чем на машинных языках.

Этапы компьютерной эры

В 1964 году корпорация Control Data Corporation объявила о создании компьютера CDC 6600. Эта занимающая целую комнату машина считается первым суперкомпьютером, и ее начальная стоимость составляла около 6 миллионов долларов США. Этот компьютер был основным вычислительным инструментом при исследованиях в ядерной физике высоких энергий. Современный смартфон превосходит его в несколько сотен раз по вычислительной мощности и объему памяти. Вдобавок он может воспроизводить видео и музыку. Причем это всего лишь телефон.

В 1964 году доминирующим языком программирования был FORTRAN, во всяком случае, в технике и науке. Языки программирования развивались не настолько бурными темпами, как оборудование, на котором они работали. Однако мир языков программирования изменился. В ходе попыток адаптации к постоянно растущим программным проектам языки обеспечили более высокую поддержку сначала структурному программированию, а затем и объектно-ориентированному программированию. Со временем не только появились новые языки, но изменились существующие.

Стандарты языка С

В настоящее время доступно множество реализаций языка С. В идеальном случае, когда вы пишете программу на С, она должна работать одинаково на любой реализации при условии, что в ней не используется код, специфичный для конкретной машины. Чтобы добиться этого на деле, различные реализации должны соответствовать общепризнанному стандарту.

Поначалу для языка С не существовало официального стандарта. С другой стороны, общепризнанным стандартом служило первое издание книги Брайана Кернигана

и Денниса Ритчи Язык программирования С (в настоящее время доступно второе издание этой книги, выпущенное издательским домом "Вильямс"); этот стандарт получил обозначение *K&R C* или Classic С (классический С). Приложение Б настоящей книги можно рассматривать в качестве руководства по реализациям языка С. Например, создатели компиляторов утверждают, что предлагают полную реализацию K&R. Однако, хотя в упомянутом приложении дано определение языка С, в нем не описана стандартная библиотека С. Язык С зависит от своей библиотеки в большей степени, нежели другие языки, поэтому возникает необходимость также и в разработке стандарта для библиотеки. При отсутствии какого-либо официального стандарта библиотека, поставляемая вместе с реализацией С для Unix, стала стандартом де-факто.

Первый стандарт ANSI/ISO C

По мере того как язык С развивался и получал все более широкое применение в различных системах, сообщество пользователей С ощутило острую потребность во всеобъемлющем, современном и строгом стандарте. Чтобы удовлетворить эту потребность, институт ANSI (American National Standards Institute – Национальный институт стандартизации США) образовал в 1983 году специальный комитет (X3J11), целью которого была разработка нового стандарта, и он формально был принят в 1989 году. Этот стандарт (ANSI С) определяет как сам язык, так и стандартную библиотеку С. Организация ISO (International Organization for Standardization – Международная организация по стандартизации) приняла стандарт языка С (ISO С) в 1990 году. По существу ISO С и ANSI С являются одним и тем же стандартом. Окончательную версию стандарта ANSI/ISO часто называют С89 (именно в этом году институт ANSI утвердил данный стандарт) или С90 (т.к. в этом году данный стандарт был утвержден ISO). Поскольку версия ANSI появилась первой, часто используется термин *ANSI C*.

Комитет X3J11 выдвинул несколько руководящих принципов. Возможно, самым интересным был принцип, гласящий: "сохраняйте дух языка С". Комитет перечислил следующие идеи, которые выступают в качестве выражений этого духа.

- Доверять программисту.
- Не препятствовать программисту делать то, что он считает необходимым.
- Не увеличивать язык и сохранять его простоту.
- Предусматривать только один способ выполнения операции.
- Делать операцию быстродействующей, даже если при этом не гарантируется переносимость.

В последнем пункте комитет имел в виду, что реализация должна определять конкретную операцию через действия, которые проявляют себя наилучшим образом на целевом компьютере, а не пытаться любой ценой навязать абстрактное универсальное определение. В ходе изучения языка вы будете сталкиваться с примерами этой философии.

Стандарт C99

В 1994 году объединенный комитет ANSI/ISO, получивший название *комитета C9X*, начал работу по пересмотру существующего стандарта, результатом которой стал стандарт С99. Комитет подтвердил базовые принципы стандарта С90, в том числе принцип малого размера и простоты языка С. Цель, озвученная комитетом, состояла в том, чтобы не добавлять в язык новые свойства за исключением тех, которые необходимы для достижения новых целей, поставленных перед языком. Одной из этих

целей была поддержка интернационализации, например, создание способов работы с наборами интернациональных символов. Второй целью была "кодификация существующих методов устранения очевидных дефектов". Таким образом, при необходимости переноса С на 64-разрядные процессоры комитет положил в основу дополнений к стандарту опыт тех, кто решал эту задачу в реальных условиях. Третьей целью было повышение пригодности языка С для выполнения критических вычислений в рамках научных и технических проектов, что делало С более привлекательной альтернативой языку FORTRAN.

Три указанных выше момента – интернационализация, исправление дефектов и повышение вычислительной полезности – были основными причинами, которые обусловили внесение изменений. Остальные планы, предусматривавшие изменения, были более консервативными по своей природе, например, минимизация несоответствий стандарту C90 и языку C++ и сохранение концептуальной простоты языка. В формулировке документа, принятого комитетом, сказано: "... комитет голосует за предоставление C++ возможности стать *большим* и амбициозным языком".

В результате изменения, внесенные в стандарт C99, позволяют сохранить естественную суть языка С, а сам язык С остается экономным, четким и эффективным. В этой книге рассматриваются многие такие изменения. Однако, поскольку стандарт несколько отличается от реальных реализаций, в настоящее время не все компиляторы полностью реализуют все изменения. Некоторые из них могут быть недоступными в конкретной системе. Либо может оказаться, что некоторые свойства C99 станут доступными только после изменения настроек компилятора.

Стандарт C11

Поддержка стандарта – процесс бесконечный, и в 2007 году комитет по стандартам приступил к созданию следующей версии стандарта, C1X, которая была выпущена как C11. Комитет выдвинут ряд новых руководящих принципов. Одним из них стало некоторое смягчение цели "доверия программисту" с учетом современной заботы о защищенности и безопасности программного кода. Комитет сделал также ряд важных наблюдений. Одно из них заключалось в том, что стандарт C99 был не настолько хорошо принят и поддержан поставщиками, как C90. В результате некоторые функциональные возможности C99 стали необязательными для C11. Одна из причин состояла в признании комитетом того, что от поставщиков, обслуживающих рынок малых компьютеров, не следует требовать поддержки функциональных возможностей, которые не используются в целевых средах. Другое наблюдение заключалось в том, что пересмотр стандарта был обусловлен не его нарушением, а потребностью следования в русле новых технологий. Один из примеров этого – добавление необязательной поддержки параллельного программирования в ответ на тенденцию применения нескольких процессоров в компьютерах. Мы кратко рассмотрим данный вопрос, но его глубокое исследование выходит за рамки данной книги.

На заметку!

В этой книге термины *ANSI C*, или в более интернациональном духе *ANSI/ISO C* либо просто *ISO C*, служат для указания функциональных возможностей, общих для C89/90 и последующих стандартов, а *C99* и *C11* — для указания новых функциональных возможностей. Иногда будут встречаться ссылки на стандарт *C90* (например, при обсуждении первого добавления того или иного свойства в язык С).

Использование языка С: семь этапов

Как уже говорилось, язык С является компилируемым языком. Если вы привыкли работать с компилируемым языком, например, с Pascal или FORTRAN, то вам известны основные действия, выполняемые для сборки программы, написанной на С. Тем не менее, если вы имели дело с интерпретируемым языком, например, BASIC, либо графическим интерфейсно-ориентированным языком, таким как Visual Basic, или если у вас вообще нет опыта программирования, тогда вы должны ознакомиться с особенностями компиляции. Мы вскоре рассмотрим этот процесс, и вы сами сможете убедиться, что он достаточно прост и практичен. Прежде всего, чтобы дать вам общее предоставление о программировании, разобьем процесс написания программы на языке С на семь этапов (рис. 1.3). Имейте в виду, что это идеализация. На практике, особенно в случае крупных проектов, вы должны перемещаться назад и вперед, используя то, чему вы научились на более позднем этапе, для уточнения результатов, которые были получены на более ранней стадии.

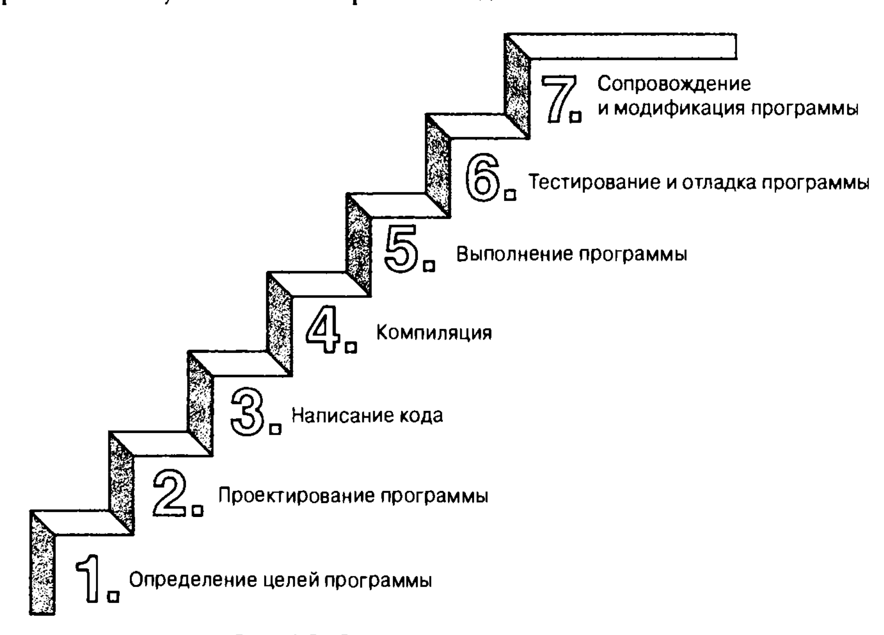


Рис. 1.3. Семь этапов программирования

Этап 1: определение целей программы

Вполне естественно, вы должны начинать с четкого представления о том, что, по вашему мнению, программа будет делать. Подумайте над тем, какая информация нужна программе, какие она должна выполнять вычисления и манипуляции, а также о том, какую информацию она должна возвращать. На этом уровне планирования следует мыслить общими понятиями, а не понятиями какого-то конкретного компьютерного языка.

Этап 2: проектирование программы

После того, как прояснения концептуальной картины того, что программа должна сделать, понадобится решить, каким образом она должна это сделать. Каким должен

быть пользовательский интерфейс? Как должна быть организована эта программа? Каковыми будут целевые пользователи? Сколько времени потребуется для завершения разработки программы?

Также необходимо решить, как представлять данные в программе и, возможно, во вспомогательных файлах, а также какие методы использовать для обработки данных. На начальном этапе изучения программирования в С ответы на эти вопросы не вызовут затруднений, но в более сложной ситуации эти решения потребуют от учета множества обстоятельств. Правильный выбор способа представления информации может существенно облегчить разработку программы и обработку данных.

Подчеркнем еще раз: нужно мыслить общими категориями и не думать о конкретном коде, однако некоторые из решений могут основываться на общих характеристиках языка. Например, программист, работающий на С, имеет гораздо больше вариантов представления данных, чем, скажем, программист, имеющий дело с языком Pascal.

Этап 3: написание кода

Теперь, при наличии четкого представления о проекте программы, можно приступать к ее реализации посредством написания кода. Иначе говоря, проект программы необходимо перевести на язык С. Именно на этой стадии потребуются все ваши знания языка С. Вы можете набросать решения на бумаге, но в конечном итоге составленный код понадобится ввести в компьютер. Механика этого процесса зависит от среды программирования, в которой вы работаете. Вскоре мы ознакомим вас с нюансами некоторых распространенных сред такого рода. В общем случае вы применяете текстовый редактор для создания так называемого файла *исходного кода*. Этот файл содержит представление проекта программы на языке С. В листинге 1.1 приведен пример исходного кода на С.

Листинг 1.1. Пример исходного кода на языке С

```
#include <stdio.h>
int main(void)
{
   int dogs;

   printf("Сколько у вас собак?\n");
   scanf("%d", &dogs);
   printf("Следовательно, у вас %d собак(а, и)!\n", dogs);

   return 0;
}
```

На этом этапе нужно документировать свои действия. Простейшим способом документирования является комментарий, которым снабжается код на С, и в который вы помещаете необходимые пояснения. В главе 2 подробно описано, как употреблять комментарии в коде.

Этап 4: компиляция

Следующим этапом разработки является компиляция исходного кода. В этом случае нюансы снова зависят от среды программирования, поэтому мы вскоре ознакомимся с рядом распространенных сред. А пока рассмотрим концептуальное представление того, что происходит на данном этапе.

Вспомните, что компилятор представляет собой программу, задачей которой является преобразование исходного кода в исполняемый код. *Исполняемый код* — это код на

собственном языке компьютера, или на *машинном языке.* Такой язык состоит из подробных инструкций, представленных в числовом коде. Как уже упоминалось, разные компьютеры имеют разные машинные языки, а компилятор языка С транслирует код С в конкретный машинный язык. Компиляторы языка С вставляют в финальную программу также код из библиотек С; эти библиотеки содержат комплект стандартных подпрограмм, например, `printf()` и `scanf()`, которыми можно пользоваться. (Если говорить точнее, то библиотечные подпрограммы в программу включает инструмент, называемый *компоновщиком* или редактором связей, но в большинстве систем его запускает компилятор.) Конечным результатом является исполняемый файл, который понимает компьютер, и который можно запускать на выполнение.

Компилятор также проверяет, не содержит ли ошибки программа на языке С. При обнаружении ошибок компилятор сообщает о них и не создает исполняемый файл. Понимание "жалоб" компилятора – это еще один навык, которым вам придется овладеть.

Этап 5: запуск программы на выполнение

Как правило, исполняемый файл представляет собой программу, которую можно запускать на выполнение. Чтобы запустить программу во многих распространенных средах, включая режим командной строки Windows, режим терминала Unix и режим терминала Linux, достаточно ввести имя исполняемого файла. Другие среды, такие как система VMS на миникомпьютерах VAX, могут потребовать ввода команды запуска или применения какого-то другого механизма. *Среды IDE* (Integrated Development Environment – интегрированная среда разработки), подобные тем, что поставляются для Windows и Macintosh, позволяют редактировать и выполнять программы на С внутри среды, выбирая соответствующие пункты меню или нажимая специальные клавиши. Полученную программу можно также запустить непосредственно из операционной системы, выполнив одиночный или двойной щелчок на имени файла или на соответствующем значке.

Этап 6: тестирование и отладка программы

То, что программа запустилась – хороший знак, тем не менее, существует вероятность, что она работает неправильно. Следовательно, необходимо убедиться, что программа делает именно то, что было задумано. Достаточно часто программы будут содержать ошибки. *Отладка* – это процесс обнаружения и исправления программных ошибок. Допущение ошибок является естественной составляющей процесса обучения. Вообще говоря, они присущи программированию, так что, сочетая программирование с обучением, лучше быть готовым к частым напоминаниям об ошибках. По мере того, как вы становитесь все более квалифицированным и искусным программистом, ваши ошибки также становятся все более масштабными и трудно обнаруживаемыми.

Есть много возможностей совершить ошибку. Можно допустить принципиальную ошибку в проекте программы. Можно некорректно реализовать хорошую идею. Вы можете упустить из виду неожиданные входные данные, которые внесут хаос в программу. Можно неправильно использовать конструкции самого языка С. Ошибки возможны при вводе кода с клавиатуры. Можно неправильно расставить скобки и т.д. Самостоятельно дополните этот список примерами из собственной практики.

К счастью, ситуация небезнадежна, хотя временами может казаться, что это именно так. Компилятор отслеживает многие виды ошибок; кроме того, можно предпринять определенные усилия, чтобы облегчить себе поиск ошибок, которые компилятор не обнаружил. По мере изучения данной книги вы найдете в ней множество советов по практической отладке программ.

Этап 7: сопровождение и модификация программы

Когда вы создаете программу для себя или кого-то другого, то, возможно, планируете ее широкое применение. Если это так, могут появиться причины для внесения в нее изменений. Вполне вероятно, что обнаружится незначительный дефект, проявляющийся при вводе имени, которое начинается с букв "Zz", либо возникнет желание улучшить что-либо в программе. Вы можете добавить в нее новую функциональную возможность. Программу можно адаптировать для выполнения в различных компьютерных системах. Решение задач подобного рода существенно упрощается, если четко документировать программу и придерживаться проверенных на практике рекомендаций.

Комментирование

Программирование обычно не является настолько прямолинейным, как описанный выше процесс. Временами приходится перемещаться вперед и назад между этапами. Например, при написании кода может выясниться, что намеченный ранее план неосуществим. Вы можете обнаружить лучший способ решения задачи или в результате анализа выполнения программы пожелать изменить проектное решение. Документирование своих действий помогает перемещаться вперед и назад между уровнями.

Многие из изучающих программирование пренебрегают этапами 1 и 2 (определение целей и проектирование программы) и переходят непосредственно к этапу 3 (написание кода). Первые написанные вами программы будут достаточно простыми, чтобы весь процесс разработки можно было "прокрутить" в голове. Если вы допустите ошибку, то найти ее будет довольно легко. По мере того как ваши программы становятся все крупнее и сложнее, представление программы в уме начинает подводить, а на выявление ошибок уходит все больше и больше времени. В конечном итоге те, кто пренебрегает стадиями планирования, обречены на бесполезную потерю времени, на путаницу и разочарование из-за громоздких, плохо функционирующих и трудных для понимания программ. Чем масштабнее и сложнее задача, тем более тщательного планирования она требует.

Мораль здесь в том, что вы должны выработать у себя привычку проводить планирование перед тем, как приступать к написанию кода. Воспользуйтесь старой, но проверенной технологией "карандаша и бумаги", чтобы сформулировать цели своей программы и набросать эскиз ее проекта. Если вы это сделаете, то в конечном итоге получите большую экономию времени и останетесь довольными результатом.

Механика программирования

Точные действия, которые нужно выполнить, чтобы получить программу, зависят от компьютерной среды. Поскольку C – переносимый язык, с ним можно работать в различных средах, включая операционные системы Unix, Linux, MS-DOS (да, некоторые все еще пользуются этой операционной системой), Windows и Macintosh. В этой книге не хватит места, чтобы рассмотреть все эти операционные среды, в частности потому, что отдельные программные продукты развиваются, умирают и заменяются другими.

Однако, прежде всего, давайте взглянем на некоторые аспекты, общие для многих сред языка C, в том числе и для указанных выше. На самом деле вы вовсе не обязаны знать, по каким правилам выполняется программа на C, но это очень полезные сведения. Они также помогают понять, почему для создания программы на C должны выполняться определенные этапы.

При написании программы на языке С код сохраняется в текстовом файле, который называется *файлом исходного кода*. Большинство систем С, в том числе упомянутые выше, требуют, чтобы имя файла заканчивалось на `.c` (например, `wordcount.c` или `budget.c`). Часть имени, находящаяся перед точкой, называется *базовым именем*, а часть, следующая за точкой — *расширением*. Таким образом, `budget` — это базовое имя, а `c` — расширение. Сочетание `budget.c` образует имя файла. Это имя должно также удовлетворять требованиям конкретной операционной системы компьютера. Например, MS-DOS представляет собой операционную систему для персональных компьютеров производства IBM и совместимых с ними. Она требует, чтобы базовое имя содержало не более восьми символов, и в силу этого обстоятельства указанное выше имя файла `wordcount.c` не является допустимым именем файла в DOS. Некоторые системы Unix ограничивают совокупную длину имени файла 14 символами, включая расширение; другие системы Unix допускают длинные имена вплоть до 255 символов. Операционные системы Linux, Windows и Macintosh также разрешают использование длинных имен.

Итак, для определенности, рассмотрим файл с именем `concrete.c`, который содержит исходный код на С, представленный в листинге 1.2.

Листинг 1.2. Программа concrete.c

```
#include <stdio.h>
int main(void)
{
   printf("Бетон содержит песок и цемент.\n");
   return 0;
}
```

Пока не беспокойтесь о деталях содержимого файла исходного кода, приведенного в листинге 1.2; мы вернемся к ним в главе 2.

Файлы объектного кода, исполняемые файлы и библиотеки

Базовая стратегия программирования на С предусматривает применение программ, которые преобразуют исходный код в исполняемый файл, содержащий готовый к выполнению код на машинном языке. Реализация программы на С обычно осуществляется в два этапа: компиляция и компоновка. Компилятор преобразует исходный код в промежуточный код, а компоновщик объединяет этот код с другим кодом, создавая исполняемый файл. В С используется такой двухэтапный подход для поддержки модульной организации программ. Индивидуальные модули можно компилировать по отдельности, а затем позже с помощью компоновщика объединять скомпилированные модули. Таким образом, если потребуется изменить какой-то один модуль, не нужно будет повторно компилировать остальные модули. Кроме того, компоновщик связывает программу с заранее скомпилированным библиотечным кодом.

Существует несколько вариантов формы промежуточных файлов. Наиболее предпочтительным является вариант, выбранный для описанных в книге реализаций, который предусматривает преобразование исходного кода в код на машинном языке, после чего результат помещается в *файл объектного кода*, или, сокращенно, *объектный файл*. (При этом предполагается, что исходный код хранится в единственном файле.) И хотя объектный файл содержит код на машинном языке, он еще не готов к запуску на выполнение. В объектном файле находится перевод исходного кода, но это еще не окончательная программа.

Первый элемент, которого не хватает в файле объектного кода – это *код запуска*, представляющий собой код, который действует в качестве интерфейса между программой и операционной системой. Например, программу можно запускать на одинаковых персональных компьютерах, один из которых функционирует под управлением Microsoft Windows, а другой – под управлением Linux. В обоих случаях оборудование одно и то же, поэтому применяется один и тот же объектный код, в то же время для Windows и для Linux нужен разный код запуска, поскольку эти системы обрабатывают программы по-разному.

Вторым отсутствующим элементом является код для библиотечных подпрограмм. Практически все программы C используют стандартные подпрограммы (называемые *функциями*), которые являются частью стандартной библиотеки C. Например, в `concrete.c` применяется функция `printf()`. Объектный файл не содержит код этой функции, в нем просто имеются команды, указывающие на использование `printf()`. Фактический код хранится в файле, который называется *библиотекой*. Библиотечный файл содержит объектный код для множества функций.

Роль компоновщика заключается в сборе вместе этих трех элементов – объектного кода, стандартного кода запуска для установленной системы и библиотечного кода – и последующем их помещении в отдельный файл, который называется исполняемым. Что касается библиотечного кода, то компоновщик извлекает только код, который необходим для функций, вызываемых из библиотеки (рис. 1.4).

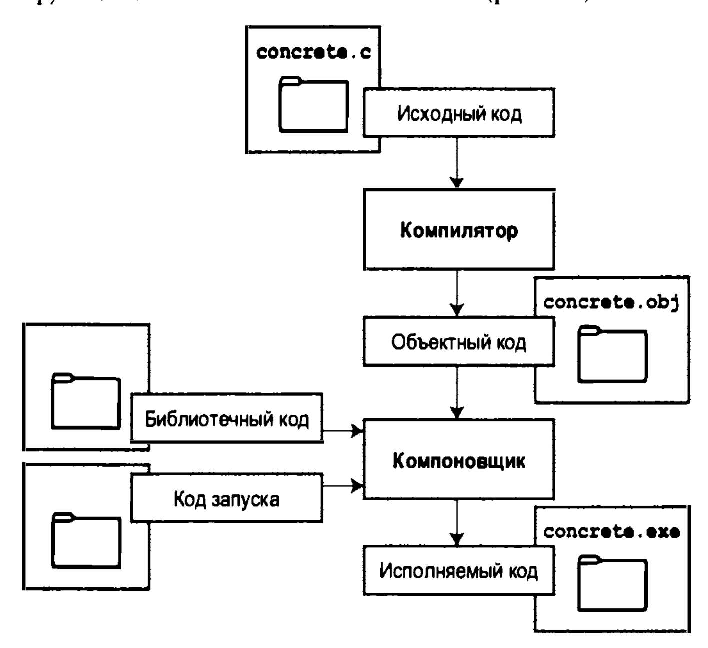


Рис. 1.4. *Компилятор и компоновщик*

Короче говоря, объектный и исполняемый файлы состоят из команд на машинном языке. Однако объектный файл содержит только результат трансляции кода программы, а исполняемый файл – также машинный код использованных стандартных библиотечных подпрограмм и код запуска.

В некоторых системах компиляцию и компоновку программ нужно запускать отдельно. В других системах компилятор запускает компоновщик автоматически, так что вам остается только выдать команду на начало компиляции.

Теперь рассмотрим несколько конкретных систем.

Операционная система Unix

Поскольку язык C появился и обрел популярность в системах Unix, мы начнем именно с этой операционной системы. (Обратите внимание: под "Unix" подразумеваются и такие системы, как FreeBSD, которая была создана на основе Unix, но не могла использовать это название по правовым причинам.)

Редактирование в системе Unix

Язык C в системе Unix не имеет собственного редактора. В этом случае применяется один из редакторов Unix общего назначения, например, emacs, jove, vi или текстовый редактор системы X Window System.

Вы отвечаете за выполнение двух процедур: корректный ввод кода программы с клавиатуры и выбор имени для файла, в котором будет храниться введенный код. Как обсуждалось ранее, это имя должно заканчиваться на `.c`. Обратите внимание, что система Unix различает прописные и строчные буквы. Поэтому `budget.c`, `BUDGET.c` и `Budget.c` — три разных допустимых имени исходных файлов, в то же время `BUDGET.C` таковым не является, т.к. в расширении `.C` используется прописная, а не строчная буква.

С помощью редактора vi мы подготовили приведенную ниже программу и сохранили ее в файле `inform.c`.

```
#include <stdio.h>
int main(void)
{
   printf("Конструкция .c завершает имя файла с программой на C.\n");
   return 0;
}
```

Приведенный текст представляет собой исходный код, а `inform.c` — исходный файл. Здесь важно отметить, что создание исходного файла — это начало процесса, но не его конец.

Компиляция в системе Unix

Наша программа, хотя и совершенна во всех других отношениях, она все же непонятна компьютеру. Компьютер не понимает таких выражений, как `#include` и `printf`. (На этой стадии, возможно, вы тоже не особо понимаете, однако у вас есть надежда вскоре узнать, что это такое, тогда как у компьютера нет никаких шансов.) Как отмечалось выше, мы нуждаемся в помощи компилятора при трансляции написанного кода (исходного кода) в код компьютера (машинный код). Результатом этих усилий будет исполняемый файл, который содержит весь машинный код, который необходим компьютеру для выполнения работы.

Исторически сложилось так, что компилятор Unix C, вызываемый командой `cc`, определил язык. Но он не шел наравне со стандартом разработки, поэтому от него отказались. Однако, как правило, системы Unix предоставляют компилятор C из какого-то другого источника, а затем превращают команду `cc` в псевдоним этого компилятора. Таким образом, можно по-прежнему использовать одну и ту же команду, хотя она и вызывает различные компиляторы в разных системах.

Чтобы скомпилировать программу `inform.c`, введите следующую команду:

```
cc inform.c
```

Спустя момент приглашение командной строки Unix отобразится снова, уведомляя о том, что дело сделано. Вы можете получить предупреждения или сообщения об ошибках, если программа написана неправильно, однако предположим, что все прошло удачно. (Если компилятор жалуется, что не понимает слова `void`, это означает, что данная система еще не имеет компилятора ANSI C. Более подробно о стандартах речь пойдет немного позже. Пока что просто удалите слово `void` из текста примера.) Если воспользоваться командой `ls` для вывода списка файлов, обнаружится новый файл с именем `a.out` (рис. 1.5). Это исполняемый файл, содержащий транслированную (или скомпилированную) программу. Чтобы запустить его, достаточно ввести

```
a.out
```

и в ответ будет выдано следующее сообщение:

```
Конструкция .c завершает имя файла с программой на C.
```

Если вы хотите сохранить исполняемый файл (`a.out`), то должны его переименовать. В противном случае он будет заменен новым файлом `a.out` при следующей компиляции программы.

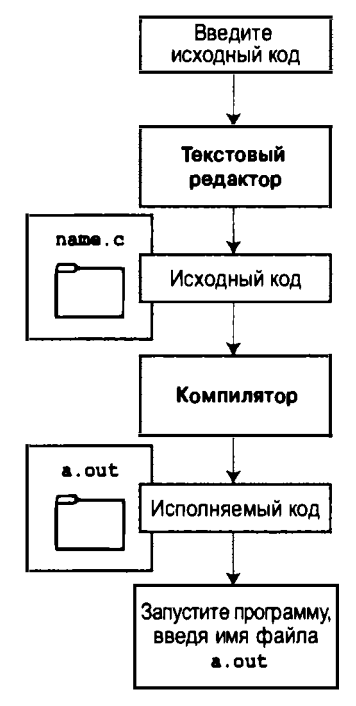


Рис. 1.5. Подготовка программы на языке C в среде Unix

А что можно сказать об объектном коде? Компилятор создает файл объектного кода, имеющий то же базовое имя, что и исходный файл, но с расширением `.o`. В нашем примере файл объектного кода получает имя `inform.o`, но вы его не найдете, поскольку компоновщик удалит его, как только построение исполняемой программы будет завершено. Однако если первоначальная программа использует более одного исходного файла, файлы объектного кода будут сохранены. При последующем рассмотрении многофайловых программ вы убедитесь, что это была здравая идея.

Коллекция компиляторов GNU и проект LLVM

Проект GNU, запущенный в 1987 году, является проектом массового сотрудничества, в рамках которого было разработано множество бесплатных Unix-подобных программ. (GNU представляет собой аббревиатуру от "GNU's Not Unix" ("GNU – это не Unix").) Одно из его детищ – коллекция компиляторов GNU, или GCC, в число которых входит и компилятор GCC для языка C. Проект GCC пребывает в состоянии постоянной разработки, которая ведется под руководством координационного комитета, и в его компиляторе C точно отслеживаются изменения стандартов языка C. Версии GCC доступны для широкого множества аппаратных платформ и операционных систем, включая Unix, Linux и Windows. Компилятор GCC C может быть вызван командой `gcc`. При этом многие системы, использующие команду `gcc`, создадут для нее псевдоним `cc`.

Проект LLVM предоставляет еще одну замену для команды `cc`. Этот проект представляет собой коллекцию связанного с компилятором программного обеспечения с открытым кодом, разработка которого началась в 2000 году с исследовательского проекта в Иллинойском Университете. Его компилятор Clang выполняет обработку кода C и может быть вызван с помощью команды `clang`. Доступный на нескольких платформах, включая Linux, в конце 2012 года Clang стал стандартным компилятором системы FreeBSD. Как и GCC, в компиляторе Clang достаточно оперативно отслеживаются изменения в стандарте C.

Оба компилятора принимают флаг `-v` для отображения информации о версии, поэтому в системах, использующих псевдоним `cc` для команды `gcc` или `clang`, следующая команда отображает сведения об используемом компиляторе и его версии:

```
cc -v
```

В зависимости от версии, как `gcc`, так и `clang` могут требовать указания параметров времени выполнения для вызова более новых стандартов C:

```
gcc -std=c99 inform.c
gcc -std=c1x inform.c
gcc -std=c11 inform.c
```

Первый пример вызывает стандарт C99, второй – черновой стандарт C11 для версий GCC, разработанных до принятия стандарта, а третий – стандарт C11 для версий GCC, которые были разработаны после его принятия. В компиляторе Clang применяются те же самые флаги.

Системы Linux

Linux является широко распространенной Unix-подобной операционной системой с открытым кодом, которая работает на различных платформах, включая PC и Mac. Подготовка программы C в среде Linux мало чем отличается от подготовки в среде системы Unix, за исключением того, что вам придется воспользоваться общедоступным и бесплатным компилятором GCC C, предоставляемым GNU. Команда компиляции имеет следующий вид:

```
gcc inform.c
```

Обратите внимание на то, что установка компилятора GCC производится по желанию пользователя во время установки системы Linux, поэтому вам (или кому-то другому) придется устанавливать компилятор GCC, если он не был установлен изначально. Как правило, при установке создается и псевдоним `cc`, указывающий на компилятор `gcc`, поэтому в командной строке можно использовать `cc` вместо `gcc`.

Дополнительная информация о GCC, включая сведения о новых версиях, доступна по адресу: `http://www.gnu.org/software/gcc/index.html`.

Компиляторы командной строки для PC

Компилятор языка С не является частью стандартного пакета Windows, поэтому может возникнуть необходимость в получении и установке этого компилятора. Cygwin и MinGW – бесплатные загружаемые файлы, которые делают компилятор GCC доступным для использования в командной строке на ПК. Cygwin запускается в собственном окне, которое выглядит подобно окну командной строки, но имитирует среду командной строки Linux. С другой стороны, MinGW выполняется в режиме командной строки Windows. Эти программы поставляются с новейшей (или почти самой новой) версией GCC, которая поддерживает стандарт C99 и, по меньшей мере, часть функциональных возможностей C11. Компилятор Borland C++ Compiler 5.5 – еще одна бесплатная загружаемая программа, которая поддерживает стандарт C90.

Файлы исходного кода должны быть текстовыми файлами, а не документами текстового процессора. (Документы текстового процессора содержат дополнительную информацию о шрифтах и форматировании.) Для работы с ними нужно применять текстовый редактор, такой как Windows Notepad. Можно воспользоваться и текстовым процессором, если с помощью пункта меню Save As (Сохранить как) сохранять файл как текстовый. Файл должен иметь расширение `.c`. Некоторые текстовые процессоры автоматически добавляют расширение `.txt` к именам текстовых файлов. Если это произойдет с вашим файлом, придется поменять его имя, заменив `txt` на `c`.

Компиляторы языка С для PC обычно, но не всегда, создают промежуточный объектный файл с расширением `.obj`. В отличие от компиляторов С для Unix, эти компиляторы, как правило, не удаляют эти файлы по завершении своей работы. Существуют компиляторы, которые генерируют файлы на языке ассемблера с расширением `.asm` либо используют собственный формат.

Некоторые компиляторы по окончании компиляции автоматически запускают компоновщик, другие могут требовать запуска компоновщика вручную. Компоновка завершается созданием исполняемого файла, при этом к первоначальному базовому имени файла исходного кода добавляется расширение `.exe`. Например, компиляция и компоновка файла исходного кода по имени `c.c` порождают файл с именем `concrete.exe`. Программу можно запустить на выполнение, введя в командной строке базовое имя файла:

```
C:>concrete
```

Интегрированные среды разработки (Windows)

Немало поставщиков, в числе которых такие компании, как Microsoft, Embarcadero и Digital Mars, предлагают среды *IDE* (integrated development environments – интегрированная среда разработки) для операционной системы Windows. (В настоящее время большинство из них представляют собой комбинированные компиляторы языков С и C++.) Бесплатные загружаемые пакеты включают Microsoft Visual Studio Express и Pelles C. Все они имеют в своем составе быстродействующие интегрированные среды, позволяющие собирать программы на языке С. Ключевой аспект в том, что каждая из этих сред имеет встроенный редактор, которым можно пользоваться для написания программ на С. Каждая IDE-среда предлагает меню, которые позволяют сохранять файлы исходного кода, а также компилировать и запускать программы, не покидая среду. Каждая IDE-среда возвращает вас обратно в редактор, если компилятор обнаруживает какие-то ошибки, при этом сопоставляя строки программы с соответствующими сообщениями об ошибках.

Среды IDE для Windows поначалу могут показаться устрашающими в силу того, что предлагают целый набор *целевых платформ*, т.е. операционных сред, в которых программа будет использоваться. Например, они могут предложить следующий выбор: 16-разрядная программа для Windows, 32-разрядная программа для Windows, файл библиотеки DLL (Dynamic-Link Library – динамически подключаемая библиотека) и т.д.

Многие целевые платформы предусматривают применение графического интерфейса Windows. Чтобы управлять этими (а также и другими) вариантами, обычно создается *проект*, куда добавляются имена файлов исходного кода, которые должны использоваться. Конкретные действия зависят от применяемого программного продукта. Как правило, сначала нужно воспользоваться меню File (Файл) или Project (Проект) для создания проекта. При этом важно выбрать правильную форму проекта. Примеры, приводимые в этой книге, носят общий характер и служат иллюстрацией выполнения программы в среде командной строки. Разнообразные IDE-среды для Windows предлагают один или несколько вариантов, чтобы соответствовать этому нетребовательному предположению.

Например, в Microsoft Visual Studio имеется вариант Win32 Console Application. В других системах ищите вариант, в котором присутствуют такие термины, как DOS EXE, Console или Character Mode executable. В этих режимах исполняемая программа будет выполняться в консольном окне. После создания проекта подходящего типа воспользуйтесь меню IDE-среды, чтобы открыть новый файл с исходным кодом. В большинстве программных продуктов это делается через меню File. Возможно, для добавления исходного файла в проект понадобится выполнить дополнительные действия.

Поскольку IDE-среды для Windows обычно рассчитаны на работу с языками C и C++, необходимо указать, что требуется создание программы на C. В некоторых интегрированных средах язык C указывается с помощью типа проекта. В других продуктах, таких как Microsoft Visual C++, для этого служит файловое расширение `.c`. В то же время большая часть программ на C работают и как программы на языке C++. Различия между языками C и C++ приведены в справочном разделе IX приложения Б.

Вы можете столкнуться с еще одной проблемой: окно, в котором отображается процесс выполнения, исчезает с экрана сразу после того, как программа завершается. В этом случае вы можете заставить программу остановиться до тех пор, пока не будет нажата клавиша <Enter>. Для этого поместите следующую строку в конец программы непосредственно перед оператором `return`:

```
getchar();
```

Эта строка считывает нажатие клавиши, поэтому программа будет ожидать нажатия клавиши <Enter>. Иногда, в зависимости от того, как функционирует программа, она уже может ожидать нажатие любой клавиши. В такой ситуации следует вызвать функцию `getchar()` два раза:

```
getchar();
getchar();
```

Например, если последнее, что сделала программа, было приглашение ввести ваш вес, вы набираете его на клавиатуре и нажимаете клавишу <Enter>, чтобы ввести эти данные. Программа считывает значение вашего веса, первый вызов функции `getchar()` прочитает нажатие клавиши <Enter>, а второй вызов `getchar()` заставит программу остановиться до тех пор, пока снова не будет нажата <Enter>. Если вы пока что не видите в этом большого смысла, то поймете сказанное после того, как освоите ввод данных в C. Позже мы еще напомним об этом подходе.

Хотя различные IDE-среды имеют много общих принципов, детали варьируются от продукта к продукту, а в рамках линейки одного продукта – от версии к версии. Вам придется немного поэкспериментировать, чтобы изучить, как работает конкретный компилятор. Кроме того, возможно, придется обратиться за советами к справочникам или поработать с онлайновым руководством.

Microsoft Visual Studio и стандарт C

Среда Microsoft Visual Studio и бесплатная версия Microsoft Visual Studio Express занимают наибольшую нишу в разработке программного обеспечения для Windows, поэтому их взаимосвязь со стандартами C весьма важна. Говоря кратко, политика Microsoft всячески поощряет программистов переходить от C к C++ или C#. Среда Visual Studio поддерживает стандарт C89/90, но ее поддержка более поздних стандартов заключается в поддержке тех новых функциональных возможностей, которые присущи также C++, таких как тип `long long`. Кроме того, начиная с версии Visual Studio 2012, среда не предлагает C в качестве одного из доступных для выбора типов проекта. Тем не менее, Visual Studio по-прежнему можно использовать с подавляющим большинством программ, описанных в этой книге. Одна из возможностей предусматривает просто выбор в настройках Application settings (Настройки приложения) опции C++, затем Win32 Console и далее Empty Project (Пустой проект). Практически все версии C совместимы с C++, поэтому большинство программ на C в этой книге также работают и как программы C++. Или же, выбрав опцию C++, для файла исходного кода можно применять расширение `.c` вместо используемого по умолчанию расширения `.cpp`, и компилятор будет работать с правилами языка C, а не C++.

Опция Windows/Linux

Многие пакеты Linux можно устанавливать из среды Windows для создания системы с двойной загрузкой. Часть дискового пространства будет выделена для системы Linux, после чего можно будет загружать либо Windows, либо Linux. Программу для Linux нельзя запускать под управлением Windows или наоборот, и к файлам Linux нельзя получать доступ из системы Windows, но к документам Windows можно обращаться из среды Linux.

Работа с языком C в системах Macintosh

В настоящее время компания Apple предоставляет свою систему разработки XCode в виде бесплатного загружаемого пакета. (В прошлом этот пакет иногда был доступен бесплатно, а иногда за умеренную плату.) Эта система позволяет работать с несколькими языками программирования, в числе которых C.

Система XCode, с ее способностями поддержки нескольких языков программирования, ориентации на множество целевых платформ и разработки крупномасштабных проектов, может казаться пугающе сложной. Но для создания простых программ на C достаточно овладеть лишь необходимым минимумом знаний. В системе XCode 4.6 воспользуйтесь меню File, чтобы выбрать опции New (Создать), Project (Проект), OS X Application Command Line Tool (Средство командной строки приложения OS X), после чего введите имя программного продукта и выберите C в качестве типа (Type). Для компиляции кода на языке C система XCode применяет компилятор Clang или GCC C. Раньше по умолчанию использовался компилятор GCC, но теперь – Clang. В настройках XCode можно указать необходимый компилятор и поддерживаемый стандарт C. (Из-за особенностей лицензирования версия Clang, доступная вместе с XCode, является более новой, чем версия GCC.)

Mac OS X построена на основе Unix, и утилита Terminal открывает окно, которое позволяет запускать программы в среде командной строки Unix. Компания Apple не предоставляет компилятор командной строки в составе своего стандартного пакета, но если загрузить XCode, можно также загрузить дополнительные инструменты командной строки, которые позволяют применять команды `clang` и `gcc` для выполнения компиляции в режиме командной строки.

Как организована эта книга

Существует много способов организации информации. Один из наиболее простых подходов заключается в том, что сначала представляется все, что касается первой темы, затем все, что имеет отношение ко второй теме, и т.д. Такой подход существенно облегчает ссылки, поскольку вы можете найти всю информацию, касающуюся данной темы, в одном месте. В то же время это не самый лучший вариант при изучении предмета. Например, если вы начнете изучать английский язык с запоминания всех существительных, то ваши возможности выражать мысли будут жестко ограничены. Разумеется, вы можете указывать на объект и выкрикивать его название, но в то же время вас будут значительно лучше понимать окружающие, если вы выучите несколько существительных, глаголов, прилагательных и прочего, а также несколько правил, указывающих, как эти элементы языка соотносятся друг с другом.

Чтобы обеспечить более рациональную подачу материала, в данной книге используется спиралевидный подход, который заключается в том, что в начальных главах начинается изучение сразу нескольких тем с возвратом к более подробному их обсуждению в последующих главах. Например, понятие функции играет важную роль в освоении языка С в целом. Таким образом, несколько начальных глав содержат краткие обсуждения функций, поэтому, когда вы приступите к чтению полного описания функций в главе 9, вам будет значительно легче осваивать тонкости применения функций. Аналогично, в начальных главах дается упрощенное предварительное описание строк и циклов, так что вы сможете пользоваться этими полезными инструментальными средствами еще до того, как вы изучите их во всех подробностях.

Соглашения, принятые в этой книге

Теперь мы готовы приступить к изучению самого языка С. В этом разделе рассматриваются некоторые соглашения, применяемые для представления материала книги.

Шрифты и начертание

Для текстов программ, входных и выходных данных используется моноширинный шрифт, который приблизительно напоминает то, что вы можете увидеть на экране или в печатном выводе. Ниже показан пример:

```
#include <stdio.h>
int main(void)
{
   printf("Бетон содержит песок и цемент.\n");
   return 0;
}
```

Тот же самый моноширинный шрифт применяется для представления терминов, связанных с кодом, например, `main()`, и имен файлов, таких как `stdio.h`.

Курсивный моноширинный шрифт используется для терминов-заполнителей, которые нужно заменять конкретными терминами, как в следующей модели объявления:

имя_типа имя_переменной;

В данном случае можно, например, вместо *имя_типа* указать `int`, а вместо *имя_переменной* — `zebra_count`.

Вывод программы

Вывод на экране компьютера представляются в том же самом формате, а входные данные пользователя выделяются полужирным начертанием. Иллюстрацией может служить следующий вывод:

```
Пожалуйста, введите название книги.
Нажмите [enter] в начале строки для останова.
Язык программирования C
Теперь введите имя автора.
Стивен Прата
```

Строки, представленные моноширинным шрифтом, являются выходными данными программы, а строка, выделенная полужирным начертанием – это данные, введенные пользователем.

Существует множество способов обмена данными между вами и компьютером. Тем не менее, мы будем полагать, что вы вводите команды с клавиатуры, а ответ компьютера читаете с экрана.

Специальные клавиши

Как правило, вы отправляете строку инструкций, нажимая клавишу, которая обозначена как <Enter>, <c/r>, <Return> или похожим образом. В тексте мы ссылаемся на нее как на клавишу <Enter>. Обычно в данной книге считается само собой разумеющимся нажатие клавиши <Enter> в конце каждой вводимой строки. Тем не менее, чтобы заострить внимание на некоторых моментах, в некоторых примерах кода клавиша <Enter> указывается явно как `[enter]`. Квадратные скобки означают, что вы нажимаете одну клавишу <Enter>, а не вводите с клавиатуры слово *enter*.

Мы также пользуемся управляющими символами, например, <Ctrl+D>. Таким способом обозначается нажатие клавиши <D> при удержании в нажатом состоянии клавиши <Ctrl> (или, возможно, <Control>).

Системы, использованные при подготовке данной книги

Некоторые аспекты языка C, такие как объем памяти, отводимый для хранения числа, зависят от системы. Когда при описании примеров мы упоминаем "наша система", обычно речь идет о компьютере iMac, работающем под управлением OS X 10.8.4 и применении системы разработки XCode 4.6.2 с компилятором Clang 3.2. Большинство программ были также скомпилированы с помощью Microsoft Visual Studio Express 2012 и Pelles C 7.0 в системе Windows 7 и GCC 4.7.3 в системе Ubuntu 13.04 Linux.

Код примеров, рассмотренных в книге, а также решения упражнений по программированию доступны для загрузки на веб-сайте издательства.

Требования к системе

Вы должны располагать компилятором С либо иметь к нему доступ. Компиляторы С имеются на огромном множестве различных компьютерных систем, так что перед вами богатый выбор. Удостоверьтесь в том, что используете компилятор С, предназначенный для вашей конкретной системы. Некоторые примеры в этой книге требуют поддержки стандарта С99 или С11, однако большинство примеров будут работать с компилятором, поддерживающим стандарт С90. Если применяемый компилятор был разработан до появления стандартов ANSI/ISO, возможно, придется достаточно часто вносить правки в код, поэтому компилятор имеет смысл обновить.

Большинство поставщиков компиляторов делают скидки для студентов и преподавателей, и если вы попадаете в эту категорию клиентов, внимательно изучите веб-сайты поставщиков.

Специальные элементы

В данной книге встречаются специальные элементы, которые подчеркивают важность того или иного вопроса. Ниже показан их внешний вид и даны пояснения, для чего они предназначены.

Врезка

Врезка содержит более глубокий анализ или дополнительную информацию, которая позволяет подробнее осветить тему.

Совет

Советы содержат краткие полезные рекомендации, касающиеся разрешения конкретных ситуаций в программировании.

Внимание!

Здесь даются предупреждения о потенциальных ловушках.

На заметку!

Нечто вроде вместилища разнообразных комментариев, которые не подпадают ни под одну из указанных выше категорий.

Резюме

С — мощный и компактный язык программирования. Его широкое распространение объясняется тем, что он предлагает полезные инструментальные средства и обеспечивает эффективное управление оборудованием, а также тем, что программы на этом языке легче переносятся с одной системы на другую.

Язык С принадлежит к числу компилируемых. Компиляторы и компоновщики (редакторы связей) языка С — это программы, которые переводят исходный код С в исполняемый код.

Программирование на языке С может требовать приложения значительных усилий, оказаться обременительным и приносить одни лишь разочарования, но в то же время оно может стать увлекательным и захватывающим занятием и доставлять только удовольствие. Мы надеемся, что язык С станет для вас источником вдохновения, каковым он стал для нас.

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля приведены в приложении А.

1. Что означает переносимость в контексте программирования?
2. Объясните, в чем состоят различия между файлом исходного кода, файлом объектного кода и исполняемым файлом.
3. Назовите семь основных этапов программирования.
4. Что делает компилятор?
5. Что делает компоновщик?

Упражнения по программированию

Мы вовсе не предполагаем, что вы уже готовы писать код на С, поэтому данное упражнение концентрируется на начальных этапах процесса программирования.

1. Вы только что были приняты на работу в компанию MacroMuscle, Inc. Компания выходит на европейский рынок и желает иметь в своем распоряжении программу, которая переводит дюймы в сантиметры (1 дюйм составляет 2,54 см). Компания хочет, чтобы программа выдавала пользователю приглашение на ввод значения в дюймах. Ваша задача заключается в том, чтобы определить цели программы и разработать проект программы (этапы 1 и 2 процесса программирования).

2

Введение в язык С

В ЭТОЙ ГЛАВЕ...

- Операция: =
- Функции: main(), printf()
- Написание простой программы на языке С
- Создание целочисленных переменных, присваивание им значений и отображение этих значений на экране
- Символ новой строки
- Включение комментариев в программы, создание программ, содержащих более одной функции, поиск ошибок в программах
- Что такое ключевые слова

На что похожа программа на языке С? Пролистав эту книгу, вы найдете множество примеров. Возможно, вы сочтете, что программа на С выглядит несколько странно, будучи усыпанной такими символами, как {, cp->tort и *ptr++. Однако по мере чтения книги, как они, так и другие характерные для С символы, уже не покажутся странными, станут более привычными и, возможно, вам даже будет трудно обходиться без них! Те читатели, которые уже знакомы с одним из множества языков, построенных на основе С, могут ощутить себя так, словно они возвратились в отчий дом к истокам детства. Эту главу мы начнем с того, что рассмотрим простую демонстрационную программу и объясним, что она делает. Одновременно мы уделим особое внимание некоторым базовым свойствам языка С.

Простой пример программы на языке С

Рассмотрим простой пример программы на языке С. Эта программа, показанная в листинге 2.1, служит для того, чтобы заострить внимание на некоторых особенностях программирования на С. Прежде чем приступать к чтению построчных пояснений к программе, ознакомьтесь с листингом 2.1 и попробуйте без помощи комментариев понять, что делает этот код.

Листинг 2.1. Программа first.c

```
#include <stdio.h>
int main(void)              /* простая программа */
{
   int num;                 /* определить переменную с именем num */
   num = 1;                 /* присвоить значение переменной num */

   printf("Я простой ");    /* использовать функцию printf() */
   printf("компьютер.\n");
   printf("Моей любимой цифрой является %d, так как она первая.\n",num);

   return 0;
}
```

Если вы думаете, что программа что-то отображает на экране, то вы не ошиблись! Однако конкретная информация, которая будет отображена на экране, может быть не очевидной, поэтому запустите программу и ознакомьтесь с ее результатами. Прежде всего, воспользуйтесь услугами своего любимого редактора (или "любимым" редактором вашего компилятора), чтобы создать файл с текстом листинга 2.1. Назначьте этому файлу имя, которое оканчивается на .c и удовлетворяет требованиям, предъявляемым к именам файлов в вашей локальной системе. Например, в качестве имени можно выбрать first.c. Теперь скомпилируйте и выполните программу. (Общие сведения по этому процессу приведены в главе 1.) Если все прошло хорошо, выходные данные программы будут иметь следующий вид:

```
Я простой компьютер.
Моей любимой цифрой является 1, так как она первая.
```

В целом результат не является неожиданным, однако что случилось с конструкциями \n и %d из программы? Кроме того, некоторые строки программы выглядят довольно странно. Самое время для пояснений.

Настройка программы

Возможно, вывод этой программы быстро мелькает на экране, а затем исчезает. Некоторые оконные среды запускают программу в отдельном окне и автоматически закрывают его после завершения программы. В таком случае в программу можно вставить дополнительный код, чтобы окно оставалось открытым до нажатия какой-либо клавиши. Один из возможных способов достижения этой цели — добавление перед оператором `return` следующей строки:

```
getchar();
```

Этот код вынуждает программу дожидаться нажатия клавиши, в результате чего окно остается открытым до ее нажатия. Функция `getchar()` более подробно описана в главе 8.

Пояснение примера

Давайте совершим два прохода по исходному коду программы. Первый проход ("Проход 1: краткий обзор") освещает значение каждой строки и поможет получить общее представление о том, что происходит. На втором проходе ("Проход 2: нюансы программы") исследуются конкретные результаты и подробности, чтобы можно было глубже понять особенности программы. На рис. 2.1 обобщены все части программы на С; на нем показано больше элементов, чем использует наша первая программа.

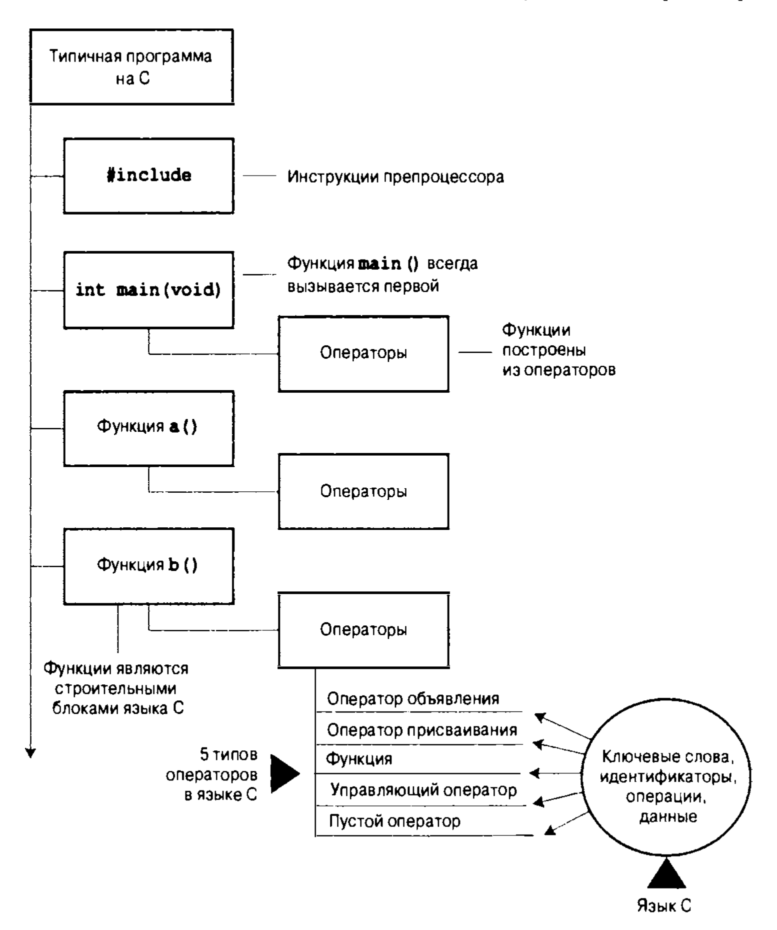


Рис. 2.1. Структура программы на языке С

Проход 1: краткий обзор

В этом разделе представлена каждая строка приведенной выше программы, за которой следует ее краткое описание; в следующем разделе тема, поднятая в этом разделе, рассматривается более подробно.

```
#include <stdio.h>           ←включить другой файл
```

Данная строка сообщает компилятору о необходимости включения информации, хранящейся в файле stdio.h, который является стандартной частью всех пакетов компилятора языка С; этот файл предоставляет поддержку клавиатурного ввода и отображения вывода.

```
int main(void)               ←имя функции
```

Программа на языке С состоит из одной или большего количества *функций* – базовых модулей любой программы С. Рассматриваемая программа состоит из одной функции по имени main. Круглые скобки идентифицируют main() как имя функции, int указывает на то, что функция main() возвращает целое число, а void – о том, что функция main() не принимает аргументов. Эти подробности будут рассмотрены позже. А сейчас просто примем int и void как часть способа определения функции main() в стандарте ANSI C. (Если в вашем распоряжении находится компилятор языка С, разработанный до появления стандарта ANSI C, удалите слово void; чтобы избежать несоответствий в дальнейшем, вам потребуется найти более новый компилятор.)

```
/* простая программа */      ←комментарий
```

Символы /* и */ заключают в себе комментарии, т.е. примечания, которые помогают понять смысл программы. Они предназначены исключительно для читателя кода и компилятором игнорируются.

```
{                            ←начало тела функции
```

Эта открывающая фигурная скобка обозначает начало оператора, образующего функцию. Определение функции заканчивается закрывающей фигурной скобкой (}).

```
int num;                     ←оператор объявления
```

Этот оператор объявляет переменную с именем num и уведомляет, что она имеет тип int (целочисленный).

```
num = 1;                     ←оператор присваивания
```

Оператор num = 1; присваивает значение 1 переменной по имени num.

```
printf("Я простой ");        ←оператор вызова функции
```

Первый оператор, использующий функцию printf(), выводит на экран текст "Я простой " и оставляет курсор в той же строке. Применяемая здесь функция printf() является частью стандартной библиотеки С. Она носит название *функции*, а использование функции в программе называется ***вызовом функции***.

```
printf("компьютер.\n");      ←еще один оператор вызова функции
```

Следующий вызов функции printf() дописывает слово "компьютер" в конец предыдущей выведенной фразы. \n – это код, указывающий компьютеру начать новую строку, т.е. переместить курсор в начало следующей строки.

```
printf("Моей любимой цифрой является %d, так как она первая.\n",num);
```

Последнее использование функции `printf()` приводит к выводу значения переменной `num` (равного 1), которое вставляется внутрь фразы, заключенной в двойные кавычки. Код `%d` указывает компьютеру, где и в какой форме вывести значение `num`.

```
return 0;    ←оператор возврата
```

Функция С может предоставить, или *возвратить*, число объекту, который ее вызвал. Пока что рассматривайте эту строку как корректный способ завершения функции `main()`.

```
}            ←конец программы
```

Как уже было сказано, программа оканчивается закрывающей фигурной скобкой.

Проход 2: нюансы программы

Теперь, когда вы вкратце ознакомились с листингом 2.1, давайте рассмотрим представленный в нем код более подробно. Мы снова будем исследовать отдельные строки программы, но на этот раз используем каждую строку кода в качестве отправной точки для более глубокого изучения нюансов, лежащих в основе кода, и как основу для того, чтобы выработать более общий взгляд на особенности программирования на С.

Директивы `#include` и заголовочные файлы

`#include <stdio.h>`

С этой строки начинается программа. Результат выполнения `#include <stdio.h>` оказывается таким же, как если бы вы ввели с клавиатуры содержимое файла `stdio.h` в своем файле там, где находится строка `#include`. В сущности, это операция вырезания и вставки. Директива `include` (включить файлы) представляет собой удобный способ совместного использования информации, который применяется во многих программах.

Оператор `#include` представляет собой пример *директивы препроцессора* в С. В общем случае компиляторы языка С выполняют некоторую подготовительную работу над исходным кодом перед компиляцией; это называется *предварительной обработкой*.

Файл `stdio.h` поставляется как часть всех пакетов компиляторов С. Он содержит информацию о функциях ввода и вывода, таких как `printf()`, и предназначен для использования компилятором. Его имя происходит от *standard input/output header* (заголовочный файл стандартного ввода-вывода). Разработчики языка С называют совокупность информации, которая помещается в верхней части файла *заголовком*, а реализации С обычно поставляются с множеством заголовочных файлов.

По большей части заголовочные файлы содержат информацию, применяемую компилятором для создания финальных исполняемых программ. Например, они могут определять константы или указывать имена функций и способы их использования. Однако фактический код функции находится в библиотечном файле предварительно скомпилированного кода, а не в заголовочном файле. Компоновщик, являющийся компонентом компилятора, позаботится о поиске необходимого библиотечного кода. Короче говоря, заголовочные файлы содействуют в правильной сборке программы.

В ISO/ANSI С стандартизировано то, какие заголовочные файлы компилятор С должен делать доступными. Для одних программ необходимо включать файл `stdio.h`, для других программ – нет. Документация по конкретной реализации языка С должна содержать описание функций из библиотеки С. Эти описания функций идентифицируют, какие заголовочные файлы нужны. Например, в описании функции `printf()` говорится о необходимости применения файла `stdio.h`. Пропуск подходящего заго-

ловочного файла может и не повлиять на какую-то конкретную программу, однако лучше на это не рассчитывать. Каждый раз, когда в приводимых здесь примерах используются библиотечные функции, будут применяться включаемые файлы, определенные стандартом ISO/ANSI для этих функций.

НА ЗАМЕТКУ! Почему ввод и вывод не являются встроенными

Может возникнуть вопрос, почему настолько базовые возможности, как ввод и вывод, не включены автоматически. Одна из причин связана с тем, что не все программы используют пакет ввода-вывода, а философия языка С запрещает перегружать программу ненужными функциями. Этот принцип экономного использования ресурсов делает язык С особо удобным для написания встроенных программ, например, кода для процессора, управляющего автоматизированной подачей топлива, или для проигрывателя Blu-ray-дисков. Кстати, строка с директивой `#include` вообще не является оператором языка С! Символ # в первой строке означает, что до передачи компилятору она должна обрабатываться препроцессором. Позже вы столкнетесь с различными примерами команд препроцессора, а в главе 16 эта тема рассматривается более подробно.

Функция `main()`

`int main(void)`

В этой строке программы объявляется функция по имени `main`. Действительно, `main` — более чем простое имя, однако это был единственно возможный выбор. Программа на языке С (с некоторыми исключениями, на которых мы сейчас не будем обращать внимание) всегда начинается с выполнения функции `main()`. Для других функций вы можете выбирать имена, однако, чтобы можно было запустить программу, в ней должна присутствовать функция `main()`. А для чего нужны скобки? Они идентифицируют `main()` как функцию. Вскоре вы узнаете больше сведений о функциях, а пока просто запомните, что функции представляют собой базовые модули программы С.

Возвращаемый тип функции `main()` определен как `int`. Это означает, что значения, которые может возвращать `main()`, являются целочисленными. Куда они возвращаются? В операционную систему — в главе 6 мы еще вернемся к этому вопросу.

В круглых скобках, которые следуют за именем функции, обычно находится информация, передаваемая функциям. В этом простом примере ничего не передается, поэтому внутри скобок находится слово `void`. (В главе 11 описан еще один формат, позволяющий передавать информацию в функцию `main()` из операционной системы.)

Просматривая старый код на С, часто можно видеть программы, которые начинаются со следующей конструкции:

```
main()
```

Стандарт С90 неохотно смирился с этой формой, а стандарты С99 и С11 ее вообще не признают. Так что даже если компилятор позволяет делать это, лучше так не поступать.

Можно также столкнуться со следующей формой:

```
void main()
```

Некоторые компиляторы разрешают такую форму, но ни в одном стандарте она не упоминается даже в качестве распознаваемого варианта. В итоге компиляторы не обязаны принимать эту форму, и некоторые из них не принимают. Поэтому придерживайтесь стандартной формы, чтобы избежать проблем при переносе программы с одного компилятора на другой.

Комментарии

/* простая программа */

Части программы, заключенные в символы /* */, представляют собой комментарии. Комментарии существенно облегчают понимание программы всеми, кто ее изучает (в том числе и вам). Одно из полезных свойств комментариев в языке C заключается в том, что они могут быть размещены в любом месте программы, даже в той же строке, где находится поясняемый код. Более длинный комментарий может располагаться в собственной строке или занимать несколько строк. Все, что находится между открывающей (/*) и закрывающей (*/) последовательностями, компилятор игнорирует. Ниже представлены примеры правильных и неправильных форм комментариев:

```
/* Это комментарий на C. */
/* Этот комментарий, будучи несколько многословным,
   размещен в двух строках. */
/*
  Допустим также и такой комментарий.
*/
/* Такой комментарий недопустим ввиду отсутствия маркера окончания.
```

В стандарте C99 появился еще один стиль комментария, который был популяризирован языками C++ и Java. Новый стиль предполагает применение символов // для представления комментария, ограниченного одной строкой:

```
// Данный комментарий умещается в одной строке.
int rigue;       // Комментарий можно также поместить сюда.
```

Поскольку конец строки означает конец комментария, этот стиль требует маркера только в начале комментария.

Новая форма комментариев решает потенциальную проблему, характерную для старой формы комментария. Предположим, что имеется следующий код:

```
/*
Я надеюсь, что этот вариант работает.
*/
x = 100;
y = 200;
/* Теперь попробуем сделать что-нибудь еще. */
```

Предположим, что вы решили удалить четвертую строку, но случайно удалили также и третью строку (*/). В результате получился такой код:

```
/*
Я надеюсь, что этот вариант работает.
y = 200;
/* Теперь попробуем сделать что-нибудь еще. */
```

Теперь компилятор соединяет в пару маркер /* из первой строки и маркер */ в четвертой строке, объединяя все четыре строки в один комментарий, в том числе и строку, которая по предположению была частью программного кода. Поскольку форма // не распространяется на более чем одну строку, не возникает проблема "исчезновения кода".

Некоторые компиляторы не поддерживают эту возможность, другие могут потребовать изменить параметры компилятора, чтобы стали доступными функции, предусмотренные стандартом C99 или C11.

Исходя из того, что чрезмерное постоянство может оказаться скучным, в книге используются обе формы комментариев.

Скобки, тела и блоки

```
{
...
}
```

В листинге 2.1 фигурные скобки определяют границы функции `main()`. В общем случае все функции языка С используют фигурные скобки для обозначения начала и конца своего тела. Наличие скобок обязательно, так что не забывайте о них. Для этой цели допускается применять только фигурные скобки (`{ }`), но не круглые (`( )`) или квадратные (`[ ]`).

Фигурные скобки можно также использовать внутри функции для организации операторов в модуль или блок. Если вам приходилось работать с языками Pascal, ADA, Modula-2 или Algol, то вы заметите, что фигурные скобки подобны операторам `begin` и `end` в упомянутых языках.

Объявления

```
int num;
```

Эта строка программы называется *оператором объявления*. Оператор объявления является одной из наиболее важных возможностей языка С. В рассматриваемом примере объявляются два аспекта. Во-первых, где-то в функции имеется *переменная* по имени `num`. Во-вторых, с помощью `int` переменная `num` объявлена как целочисленная, т.е. число без десятичной точки, или без дробной части. (`int` представляет собой пример *типа данных*.) Компилятор применяет эту информацию для того, чтобы выделить в памяти для переменной `num` пространство подходящего размера. Точка с запятой в конце строки показывает, что данная строка является *оператором* или инструкцией языка С. Точка с запятой является частью этого оператора, а не просто разделителем между операторами, как, например, в языке Pascal.

Слово `int` представляет собой *ключевое слово* языка С, обозначающее один из базовых типов данных С. Ключевые слова — это слова, используемые для построения языковых конструкций, и их нельзя употреблять в других целях. Например, `int` нельзя применять в качестве имени функции или переменной. Однако эти ограничения по использованию ключевых слов не выходят за рамки языка, так что вы вполне можете дать своему домашнему питомцу кличку *int*.

Слово `num` в данном примере является *идентификатором*, т.е. именем, которое вы выбираете для переменной, функции или другой сущности. Таким образом, объявление соединяет конкретный идентификатор с конкретной ячейкой в памяти компьютера и при этом устанавливает тип информации, или тип данных, которые будут там храниться.

В языке С *все* переменные должны быть объявлены *до* того, как они будут использоваться. Это значит, что нужно предоставить списки всех переменных, применяемых в программе, и указать, к какому типу данных принадлежит каждая переменная. Объявление переменных считается хорошим тоном в программирования, и в языке С оно обязательно.

По традиции язык С требует, чтобы переменные были объявлены в начале блока, причем объявлениям не должны предшествовать какие-то другие операторы. То есть тело функции `main()` может иметь следующий вид:

```
int main() // традиционные правила
{
    int doors;
```

```
    int dogs;
    doors = 5;
    dogs = 3;
    // другие операторы
}
```

Следуя обычаю языка C++, стандарты C99 и C99 позволяют размещать объявления в любом месте блока. Тем не менее, вы по-прежнему должны объявлять переменную до ее первого использования. Поэтому если ваш компилятор поддерживает эту возможность, код может выглядеть так:

```
int main()          // действующие в настоящее время правила C
{
    // какие-то операторы
    int doors;
    doors = 5;      // первое использование переменной doors
    // еще какие-то операторы
    int dogs;
    dogs = 3;       // первое использование переменной dogs
    // другие операторы
}
```

В целях лучшей совместимости с более ранними системами в книге мы будем придерживаться первоначальных соглашений.

Вполне возможно, что у вас возникли три вопроса. Во-первых, что такое тип данных? Во-вторых, какие есть варианты при выборе имени? В-третьих, а почему вообще нужно объявлять переменные? Давайте посмотрим, как выглядят ответы.

Типы данных

В языке C доступно несколько видов (или типов) данных: например, целые числа, символы и числа с плавающей запятой. Объявление переменной как имеющей целочисленный или символьный тип позволяет компьютеру должным образом хранить, осуществлять выборку и интерпретировать данные. В следующей главе вы ознакомитесь со всем разнообразием доступных типов.

Выбор имени

Для переменных следует выбирать осмысленные имена (или идентификаторы), например, `sheep_count` вместо `x3`, если программа занимается подсчетом овец. Если имен недостаточно, добавьте комментарии с объяснениями того, какие данные эти переменные представляют. Документирование программы в подобное манере считается хорошим тоном в программировании.

Стандарты C99 и C11 разрешают использовать имена идентификаторов любой желаемой длины, но компилятор должен рассматривать в качестве значащих только первые 63 символа. В случае внешних идентификаторов (глава 12) распознаваться будут только 31 символ. Это заметное увеличение по сравнению с требованиями стандарта C90, составляющими 31 и 6 символов, соответственно, а более старые компиляторы часто останавливались на максимум 8 символах. В действительности можно использовать больше символов, чем указанный максимум, но компилятор просто не обязан обращать внимание на дополнительные символы. Что это значит? При наличии двух идентификаторов длиной по 63 символа, отличающихся только одним символом, компилятор должен распознать их как разные идентификаторы. Если же два идентификатора длиной по 64 символа имеют отличие только в последнем символе, то компилятор может распознать их как разные, а может и не распознать; в стандарте ничего не определено относительно того, что должно происходить в таком случае.

В вашем распоряжении имеются буквы нижнего и верхнего регистров, цифры и знак подчеркивания (_). Первым символом должна быть буква или знак подчеркивания. Ниже приведены примеры допустимых и недопустимых имен.

Допустимые имена	Недопустимые имена
`wiggles`	`$Z]**`
`cat2`	`2cat`
`Hot_Tub`	`Hot-Tub`
`taxRate`	`tax rate`
`_kcab`	`don't`

В операционных системах и в библиотеке C часто применяются идентификаторы, начинающиеся с одного или двух символов подчеркивания, например, `_kcab`, поэтому лучше избегать использования таких имен в своем коде. Стандартные идентификаторы, имеющие в начале один или два символа подчеркивания, такие как библиотечные идентификаторы, являются *зарезервированными*. Это означает, что хотя их применение не вызывает синтаксической ошибки, оно может привести к конфликту имен.

В C имена *чувствительны к регистру символов*, т.е. прописная буква рассматривается как отличающаяся от соответствующей строчной буквы. Таким образом, идентификатор `stars` отличается от `Stars` и `STARS`.

Чтобы придать языку C более высокую интернациональность, стандарты C99 и C99 обеспечивают доступность обширного набора символов посредством механизма UCN (Universal Character Names — имена в универсальных символах). Подробное описание этого расширения приведено в приложении Б. Эта возможность позволяет использовать символы, не входящие в английский алфавит.

Четыре веских причины объявления переменных

Некоторые ранние языки программирования, такие как первоначальные формы FORTRAN и BASIC, позволяли применять переменные без их объявления. А почему нельзя использовать такой упрощенный подход в C? Это обусловлено рядом причин.

- Размещение объявлений всех переменных в одном месте упрощает читателю кода уловить назначение программы. Это особенно справедливо, когда вы назначаете переменным осмысленные имена (например, `taxrate` вместо `r`). Если имени недостаточно, предусмотрите в комментарии объяснение, что конкретно представляют объявленные переменные. Документирование программы в таком стиле считается хорошим тоном в программировании.
- Обдумывание того, какие переменные объявить, способствует проведению определенного планирования, прежде чем приступать к написанию кода. Какая информация нужна для того, чтобы начать писать программу? Какой вывод должна производить программа? Как лучше всего представить данные?
- Объявление переменных помогает избежать одной из наиболее тонких и трудных для обнаружения ошибок программирования — некорректно написанного имени переменной.

  Предположим, что на одном из языков с необязательным объявлением переменных был написан следующий оператор:

  ```
  RADIUS1 = 20.4;
  ```

Затем в другом месте программы был введен оператор с неправильно указанным именем переменной:

```
CIRCUM = 6.28 * RADIUSl;
```

Вы не заметили, как вместо цифры 1 ввели букву l (строчную латинскую букву "L"). Согласно правилам этого языка будет создана новая переменная с именем RADIUSl, которая получит случайное значение (возможно ноль, а возможно какой-то мусор). Переменная CIRCUM получит неправильное значение, и придется потратить немало времени, чтобы выяснить причину. В С это невозможно (если только вы не окажетесь достаточно неосмотрительными, объявив два настолько похожих имени), т.к. компилятор выдаст сообщение об ошибке, когда в коде встретится необъявленная переменная RADIUSl.

- Пока вы не объявите переменные, программа на С не скомпилируется. Если перечисленные выше причины не возымели действия, то этот серьезный аргумент должен окончательно убедить.

Учитывая необходимость объявления переменных, где это следует делать? Как упоминалось ранее, до появления стандарта C99 требовалось размещать все объявления в начале блока. Одна из причин следования этой рекомендации заключается в том, что группирование объявлений в едином месте облегчает понимание назначения программы. Разумеется, существуют аргументы и в пользу распределения объявлений по всей программе, как теперь разрешает делать стандарт C99. Идея в том, чтобы объявлять переменные непосредственно перед тем, когда вы готовы присвоить им значения. Это позволит не забыть присвоить переменным начальные значения. Однако на деле многие компиляторы пока не поддерживают такое правило стандарта C99.

Присваивание

```
num = 1;
```

В следующей строке программы находится *оператор присваивания*, в котором применяется одна из основных операций языка С. В рассматриваемом примере это означает "присвоить значение 1 переменной num". Предшествующая ему строка int num; резервирует в памяти компьютера пространство для переменной num, а строка с оператором присваивания сохраняет значение в этой ячейке. Позже при желании переменной num можно присвоить другое значение; вот почему num называется переменной. Обратите внимание, что оператор присваивания назначает значение, указанное справа знака операции, переменной, указанной слева. Кроме того, оператор завершается точкой с запятой (рис. 2.2).

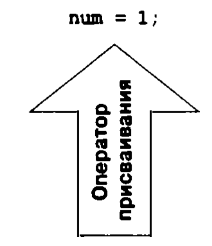


Рис. 2.2. Оператор присваивания является одной из базовых операций в С

Функция printf()

```
printf("Я простой ");
printf("компьютер.\n");
printf("Моей любимой цифрой является %d, так как она первая.\n",num);
```

Во всех этих строках используется стандартная функция С по имени printf(). Круглые скобки указывают, что printf является именем функции. То, что содержится внутри круглых скобок – это информация, передаваемая из функции main() в функцию printf(). Например, первая строка передает фразу "Я простой " в функцию printf(). Такая информация называется *аргументом* или, более точно, *фактическим*

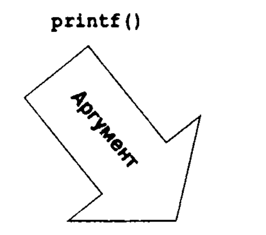


Рис. 2.3. Функция printf() *с аргументом*

аргументом функции (рис. 2.3). (Для различения конкретного значения, переданного функции, и переменной в функции, используемой для хранения значения, в языке С используются термины *фактический аргумент* и *формальный аргумент.* Более подробно об этом пойдет речь в главе 5.) Что делает функция printf() с этим аргументом? Она просматривает все, что заключено в двойные кавычки, и выводит этот текст на экран.

Первая строка с printf() является примером того, как *вызвать* или как *обратиться* к функции в С. Понадобится только ввести имя функции и поместить нужный аргумент (аргументы) в круглые скобки. Когда выполнение достигает этой строки, управление передается указанной функции (printf() в рассматриваемом случае). После того как функция выполнит свою работу, управление возвращается в исходную (*вызывающую*) функцию – main() в данном примере.

Чем отличается следующая строка с printf()? Она содержит символы \n, заключенные в кавычки, и они не выводятся! В чем же дело? Символы \n означают начало новой строки. Комбинация \n (вводится как два символа) представляет один символ, получивший название *символа новой строки.* Для функции printf() эта комбинация означает "начать новую строку с крайней левой позиции". Другими словами, вывод символа новой строки выполняет ту же операцию, что и нажатие клавиши <Enter> на стандартной клавиатуре. А почему бы просто не нажать клавишу <Enter> при наборе этого аргумента printf()? Потому что это будет воспринято как непосредственная команда редактору, но не как инструкция, которая должна быть помещена в исходном коде. В результате при нажатии клавиши <Enter> редактор прейдет с текущей строки на следующую. Тем не менее, символ новой строки влияет на то, как будет отображаться вывод программы.

Символ новой строки является примером *управляющей последовательности.* Управляющая последовательность применяется для представления символов, которые трудно или просто невозможно ввести с клавиатуры. Примерами таких последовательностей могут служить символы \t для представления нажатия клавиши <Tab> и \b – для <Backspace>. В любом случае управляющая последовательность начинается с обратной косой черты (\). Мы вернемся к этой теме в главе 3.

Таким образом, все это объясняет, почему три оператора printf() вывели только две строки: первый оператор не содержал символа новой строки, но он был включен во второй и третий операторы.

Финальная строка с printf() привносит еще одну странность: что случилось с %d при выводе строки? Вспомните, что вывод этой строки выглядел так:

```
Моей любимой цифрой является 1, так как она первая.
```

Итак, при выводе этой строки вместо группы символов %d появилась цифра 1, и 1 – это значение переменной num. Комбинация %d представляет собой заполнитель, который показывает, где должно быть выведено значение переменной num. Эта строка подобна следующему оператору BASIC:

```
PRINT "Моей любимой цифрой является "; num; ", так как она первая."
```

Версия С фактически делает немного больше. Символ % уведомляет программу, что в этом месте будет выведено значение переменной, а d указывает на то, что перемен-

ная должна выводиться как десятичное целое число. Функция `printf()` предлагает на выбор несколько вариантов, включая шестнадцатеричные целые числа и числа с плавающей запятой. Действительно, буква `f` в имени `printf()` является напоминанием о том, что это *форматирующая* функция вывода. Каждый тип данных имеет собственный спецификатор; по мере того, как в данной книге будут вводиться все новые типы, будут также представлены и соответствующие спецификаторы.

Оператор возврата

```
return 0;
```

Оператор возврата является завершающим оператором программы. `int` в конструкции `int main(void)` означает, что функция `main()` возвращает целочисленное значение. Стандарт языка С требует, чтобы поведение функции `main()` было именно таким. Функции С, возвращающие значения, делают это с помощью оператора возврата, состоящего из ключевого слова `return`, за которым следует возвращаемое значение и точка с запятой. Если в функции `main()` опустить оператор возврата, по достижении закрывающей фигурной скобки `}` программа возвратит значение `0`. Таким образом, оператор возврата в конце функции `main()` можно не указывать. Однако для других функций это не разрешено, поэтому ради единообразия рекомендуем использовать оператор возврата также и в `main()`. На этом этапе вы можете считать оператор возврата в функции `main()` чем-то необходимым для обеспечения логической согласованности, но в некоторых операционных системах, включая Linux и Unix, он имеет практическое применение. В главе 11 эта тема рассматривается более подробно.

Структура простой программы

Теперь, когда вы видели конкретный пример, вы готовы к ознакомлению с несколькими общими правилами для программ на С. *Программа* состоит из коллекции одной или нескольких функций, одна из которых обязательно должна иметь имя `main()`. Описание *функции* включает заголовок и тело функции. *Заголовок функции* содержит имя функции и сведения о типе информации, передаваемой в функцию и возвращаемой из нее. Имя функции можно опознать по круглым скобкам, которые могут быть пустыми. *Тело* функции заключено в фигурные скобки (`{}`) и состоит из последовательности операторов, каждый из которых завершается точкой с запятой (рис. 2.4). В примере, приведенном в настоящей главе, использовался *оператор объявления*, определяющий имя и тип переменной. В нем также присутствовал *оператор присваивания*, устанавливающий значение переменной. Кроме того, в нем применялись три *оператора вывода*, в каждом из которых вызывалась функция `printf()`. Эти операторы вывода представляют собой примеры *операторов вызова функции*. И, наконец, функция `main()` завершается *оператором возврата*.

Короче говоря, простая стандартная программа на С должна иметь следующий формат:

```
#include <stdio.h>
int main(void)
{
   операторы
   return 0;
}
```

(Помните, что каждый оператор завершается символом точки с запятой.)

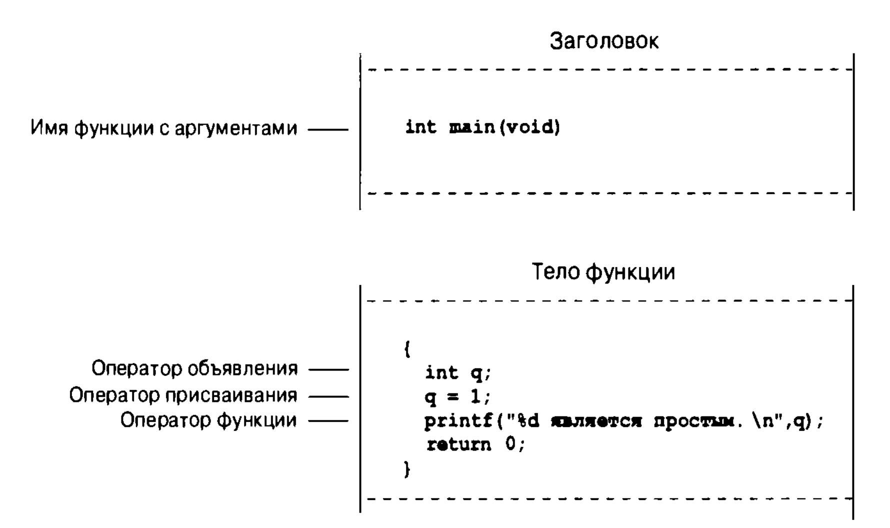


Рис. 2.4. Функция имеет заголовок и тело

Советы по обеспечению читабельности программ

Написание удобочитаемых программ является хорошим тоном в программировании. Удобочитаемую программу легче понять, и ее проще корректировать или модифицировать. Процесс придания программе удобочитаемого вида также помогает прояснить собственную концепцию того, что делает программа.

Вы уже видели два приема улучшения читабельности: выбор осмысленных имен для переменных и использование комментариев. Обратите внимание на то, что оба эти приема дополняют друг друга. Если назначить переменной имя `width` (ширина), то в комментарии с пояснением, что эта переменная представляет ширину, нет необходимости; в то же время переменная по имени `video_routine_4` (видеопрограмма 4) требует объяснения, для чего она предназначена.

Существует также прием, предусматривающий применение пустых строк для отделения одного концептуального раздела функции от другого. Например, в простой демонстрационной программе присутствует пустая строка, отделяющая раздел объявлений от раздела действий. В коде на С пустые строки не обязательны, в то же время они повышают читабельность программы.

Четвертый прием заключается в размещении каждого оператора в одной строке. Опять-таки, это вопрос соглашения о читабельности, а не требование языка С. В языке С принят формат *свободной формы*. Вы можете помещать несколько операторов в одну строку или разносить одиночный оператор на множество строк. Приведенный ниже код допустим, хоть и неуклюж:

```
int main( void ) { int four; four
=
4
;
printf(
     "%d\n",
four); return 0;}
```

Точка с запятой сообщает компилятору, где заканчивается один оператор и начинается другой, но логика программы будет намного яснее, если следуете соглашениям, используемым в примере настоящей главы (рис. 2.5).

```
int main(void) /* преобразует две морских сажени в футы */

{
  int feet, fathoms;

  fathoms = 2;
  feet = 6 * fathoms;
  printf("В %d морских саженях содержится %d футов!\n", fathoms, feet);
  return 0;
}
```


Рис. 2.5. Придание программе удобочитаемого вида

Еще один шаг в использовании языка С

Первая демонстрационная программа была совсем простой, и следующий пример, представленный в листинге 2.2, не намного труднее.

Листинг 2.2. Программа fathm_ft.c

```
// fathm_ft.c -- преобразует две морских сажени в футы
#include <stdio.h>
int main(void)
{
    int feet, fathoms;

    fathoms = 2;
    feet = 6 * fathoms;
    printf("В %d морских саженях содержится %d футов!\n", fathoms, feet);
    printf("Да, именно %d футов!\n", 6 * fathoms);

    return 0;
}
```

Что здесь нового? В этом коде предоставлено описание программы, объявлено несколько переменных, выполнено умножение и выведены на экран значения двух переменных. Давайте рассмотрим все это более подробно.

Документирование

Во-первых, программа начинается с комментария (с применением нового стиля), идентифицирующего имя файла программы и назначение программы. Такой вид документирования не требует много времени, но принесет большую пользу позднее, когда вы будете просматривать множество файлов или печатать их содержимое.

Множественные объявления

Во-вторых, в программе объявлены сразу две переменные без использования отдельного оператора объявления для каждой переменной. Для этого в операторе объявления переменные (feet и fathoms) должны разделяться запятыми. Это значит, что

```
int feet, fathoms;
```

и

```
int feet;
int fathoms;
```

эквивалентны.

Умножение

В-третьих, в программе выполняется умножение. Она использует огромную вычислительную мощь компьютерной системы для умножения 2 на 6. В C, как и во многих языках программирования, символом умножения является *. Таким образом, оператор

```
feet = 6 * fathoms;
```

означает "получить значение переменной fathoms, умножить его на 6 и присвоить результат вычисления переменной feet".

Вывод нескольких значений

Наконец, в-четвертых, в этой программе довольно необычно применяется функция printf(). Если скомпилировать и выполнить этот пример, вывод будет выглядеть примерно так:

```
В 2 морских саженях содержится 12 футов!
Да, именно 12 футов!
```

На этот раз в первом вызове printf() сделано *две* подстановки. Первая комбинация %d в кавычках была заменена значением первой переменной (feet) в списке, который следует за сегментом в кавычках, а вторая такая комбинация заменена значением второй переменной (fathoms) из этого списка. Обратите внимание, что список переменных, предназначенных для вывода, находится в хвостовой части этого оператора после части, заключенной в кавычки. Также следует отметить, что каждый элемент списка отделяется от других запятой.

Второй случай использования printf() демонстрирует тот факт, что выводимое значение не обязательно должно быть переменной; оно вполне может быть чем-то вроде выражения 6 * fathoms, которое приводится к соответствующему типу.

Эта программа имеет ограниченную область применения, однако она может служить ядром программы для преобразования морских саженей в футы. Все что для этого понадобится — способ интерактивного присваивания дополнительных значений переменной feet; далее в этой главе будет показано, как это делать.

Множество функций

До сих пор в программах использовалась стандартная функция printf(). В листинге 2.3 показано, как можно внедрить в программу собственную функцию помимо main().

Листинг 2.3. Программа two_func.c

```
/* two_func.c -- программа, в которой используются две функции в одном файле */
#include <stdio.h>
void butler(void);     /* прототип функции в стандарте ISO/ANSI C */
int main(void)
{
   printf("Я вызываю дворецкого.\n");
   butler();
   printf("Да. Принесите мне чай и записываемые DVD-диски.\n");
   return 0;
}
void butler(void) /* начало определения функции */
{
   printf("Вы звонили, сэр?\n");
}
```

Вывод будет иметь следующий вид:

```
Я вызываю дворецкого.
Вы звонили, сэр?
Да. Принесите мне чай и записываемые DVD-диски.
```

Функция butler() встречается в этой программе трижды. В первый раз она появляется в виде *прототипа*, который информирует компилятор о функциях, которые будут применяться. Во второй раз она присутствует внутри main() в форме *вызова функции*. В третий раз в программе представлено *определение функции*, которое является исходным кодом самой функции. Рассмотрим по очереди каждое из этих трех появлений.

Прототипы были добавлены в стандарте C90, поэтому более старые компиляторы их не распознают. (Вскоре будет дано объяснение, что делать, когда приходится работать с такими компиляторами.) Прототип объявляет компилятору о том, что вы применяете конкретную функцию, поэтому он и называется *объявлением функции*. Он также определяет свойства этой функции. Например, первое ключевое слово void в прототипе для функции butler() указывает на то, что butler() не имеет возвращаемого значения. (В общем случае функция может возвращать значение в вызывающую функцию для последующего его использования, но в случае butler() это не так.) Второе void – то, которое в butler(void) – означает, что функция butler() не принимает аргументов. Поэтому когда компилятор достигает места в main(), где вызывается butler(), он может проверить, корректно ли применяется эта функция. Обратите внимание, что ключевое слово void употребляется в смысле "пусто", а не в смысле "недействительно".

В ранних версиях C поддерживалась более ограниченная форма объявления функции, в которой можно было определять только возвращаемый тип, опуская при этом описание аргументов:

```
void butler();
```

В старом коде на C вместо прототипов функций использовались объявления функций, подобные показанному выше. Стандарты C90, C99 и C11 распознают такую устаревшую форму, но также указывают на то, что со временем от нее откажутся, поэтому ее лучше не применять. Если вы имеете дело с унаследованным кодом, имеет смысл привести объявления старого типа к прототипам. В последующих главах мы еще вернемся к рассмотрению прототипов, объявлений функций и возвращаемых значений.

Далее функция butler() вызывается внутри main() путем указания ее имени и круглых скобок. Когда функция butler() завершит свою работу, управление переходит на следующий оператор внутри main().

Наконец, функция butler() определена точно так же, как main(), с заголовком и телом, заключенным в фигурные скобки. Заголовок повторяет информацию, указанную в прототипе: функция butler() не принимает аргументов и ничего не возвращает. Для компиляторов ранних версий второе вхождение void понадобится удалить.

Следует отметить еще один момент: фактическое выполнение функции butler() зависит не от места ее определения в файле, а от места вызова butler() внутри main(). Например, в приведенной выше программе определение функции butler() можно было бы поместить перед определением main(), и программа вела бы себя точно так же, выполняя функцию butler() между двумя вызовами printf() внутри main(). Вспомните, что все программы на С начинают выполнение с функции main(), при этом не имеет значения, в каком месте файла исходного кода эта функция находится. Однако обычной практикой является определение функции main() первой, поскольку это позволяет получить представление о базовой инфраструктуре программы.

Стандарт языка С рекомендует предоставлять прототипы для всех используемых функций. Стандартные файлы include позаботятся о решении этой задачи для стандартных библиотечных функций. Например, согласно стандарту языка С, файл stdio.h содержит прототип функции printf(). В заключительном примере главы 6 демонстрируется способ распространения прототипов на функции не void, а в главе 9 такие функции рассматриваются более подробно.

Знакомство с отладкой

Теперь, когда вы знаете, как написать простую программу на С, возникает шанс появления простых ошибок. Поиск и исправление ошибок называется *отладкой*. В листинге 2.4 представлена программа с несколькими ошибками. Посмотрите, сколько их вы сумели обнаружить.

Листинг 2.4. Программа nogood.c

```
/* nogood.c -- программа с ошибками */
#include <stdio.h>
int main(void)
(
   int n, int n2, int n3;

   /* В этой программе допущено несколько ошибок
   n = 5;
   n2 = n * n;
   n3 = n2 * n2;
   printf("n = %d, n в квадрате = %d, n в кубе = %d\n", n, n2, n3)

   return 0;
)
```

Синтаксические ошибки

Код в листинге 2.4 содержит ряд синтаксических ошибок. *Синтаксическая ошибка* возникает в случае нарушения правил языка С. Она аналогична грамматической ошибке в обычном тексте.

В качестве примера рассмотрим предложение "Быть программные ошибки катастрофическими могут". В этом предложении используются правильные слова, однако порядок их следования некорректен. Синтаксические ошибки в С связаны с тем, что допустимые символы языка размещаются в неправильных местах.

Итак, какие синтаксические ошибки присутствуют в программе nogood.c? Во-первых, для пометки тела функции применены круглые скобки, а не фигурные, т.е. допустимые в целом символы языка С находятся в неподходящем месте. Во-вторых, объявления должны выглядеть так:

```
int n, n2, n3;
```

или, возможно, так:

```
int n;
int n2;
int n3;
```

Далее, в коде отсутствует пара символов */, необходимая для завершения комментария. (Пару символов /* можно было бы заменить новой формой комментария //.) Наконец, в коде пропущен обязательный символ точки с запятой, который должен завершать оператор printf().

Как обнаруживать синтаксические ошибки? Во-первых, перед компиляцией нужно просмотреть исходный код — возможно, удастся найти какие-то очевидные ошибки. Во-вторых, можно исследовать ошибки, найденные компилятором, т.к. одной из его задач является именно выявление синтаксических ошибок. При попытке компиляции такой программы компилятор сообщает об обнаруженных им ошибках, идентифицируя природу и местоположение каждой ошибки.

Однако и сам компилятор может ошибаться. Настоящая синтаксическая ошибка в одном месте может ввести компилятор в заблуждение и заставить его предполагать, что он нашел другие ошибки. Например, поскольку в примере неправильно объявлены переменные n2 и n3, компилятор может посчитать, что он обнаружил еще несколько ошибок там, где используются эти переменные. В действительности, если не удается разобраться во всех обнаруженных ошибках, то вместо того, чтобы пытаться исправлять сразу все ошибки, сначала следует исправить первые одну или две ошибки, после чего выполнить повторную компиляцию; вполне возможно, что какие-то другие ошибки исчезнут. Продолжайте в том же духе, пока программа не заработает. Еще одна распространенная особенность компилятора заключается в том, что он сообщает об ошибке на одну строку позже. Например, компилятор может не догадаться, что не хватает точки с запятой, пока не наступит очередь компиляции следующей строки. Таким образом, если компилятор жалуется на отсутствие точки с запятой в строке, в которой этот символ имеется, проверьте предыдущую строку.

Семантические ошибки

Семантические ошибки — это смысловые ошибки. В качестве примера рассмотрим следующее предложение: "Презрительные наследники напевают зелено". Синтаксических ошибок оно не содержит, т.к. прилагательное, существительное, глагол и наречие находятся на своих местах, тем не менее, само предложение бессмысленно. В языке С семантическая ошибка возникает, когда вы соблюдаете все требования языка, но получаете некорректный результат. В рассматриваемом примере присутствует одна такая ошибка:

```
n3 = n2 * n2;
```

В данном случае предполагается, что n3 представляет куб числа n, в то время как код вычисляет четвертую степень n.

Компилятор не обнаруживает семантических ошибок, поскольку они не нарушают правила языка С. Компилятор не способен предугадывать ваши истинные намерения. Поэтому искать ошибки такого рода придется самостоятельно. Один из способов предусматривает сравнение того, что программа делает, с тем, что вы хотели от нее получить. Например, предположим, что вы исправили синтаксические ошибки в рассматриваемом примере, так что код теперь приобрел вид, представленный в листинге 2.5.

Листинг 2.5. Программа stillbad.c

```
/* stillbad.c -- программа с устраненными синтаксическими ошибками */
#include <stdio.h>
int main(void)
{
   int n, n2, n3;

   /* В этой программе есть семантическая ошибка */
   n = 5;
   n2 = n * n;
   n3 = n2 * n2;
   printf("n = %d, n в квадрате = %d, n в кубе = %d\n", n, n2, n3);

   return 0;
}
```

Вывод программы выглядит следующим образом:

```
n = 5, n в квадрате = 25, n в кубе = 625
```

Несложно заметить, что 625 не является правильным результатом возведения числа в третью степень. Следующий этап предусматривает отслеживание того, как был получен такой результат. В рассматриваемом примере ошибку, скорее всего, удастся выявить путем инспекции кода. Однако в общем случае нужно применять более систематизированный подход. Один из методов — пошаговое отслеживание инструкций программы. Воспользуемся этим методом и в данном случае.

Тело программы начинается с объявления трех переменных: n, n2, и n3. Эту ситуацию можно смоделировать, нарисовав три прямоугольника и пометив их именами переменных (рис. 2.6). Далее программа присваивает переменной n значение 5. Смоделируйте это действие, записав 5 в прямоугольник n. Затем программа умножает n на n и присваивает результат переменной n2. Посмотрев в прямоугольник n, вы увидите, что в нем находится значение 5. Умножьте 5 на 5 и получите 25, после чего поместите 25 в прямоугольник n2. Чтобы воспроизвести следующий оператор С (n3 = n2 * n2;), загляните в прямоугольник n2; вы там найдете 25. Умножьте 25 на 25, получите 625 и поместите это значение в прямоугольник n3. Итак, вы возводите n2 в квадрат вместо того, чтобы умножить его на n.

Возможно, эта процедура и избыточна для данного простого примера, однако подобного рода пошаговое выполнение программы часто является наилучшим способом посмотреть, что в ней происходит.

Состояние программы

Выполняя пошаговый просмотр программы вручную с отслеживанием каждой переменной, вы осуществляете мониторинг состояния программы.

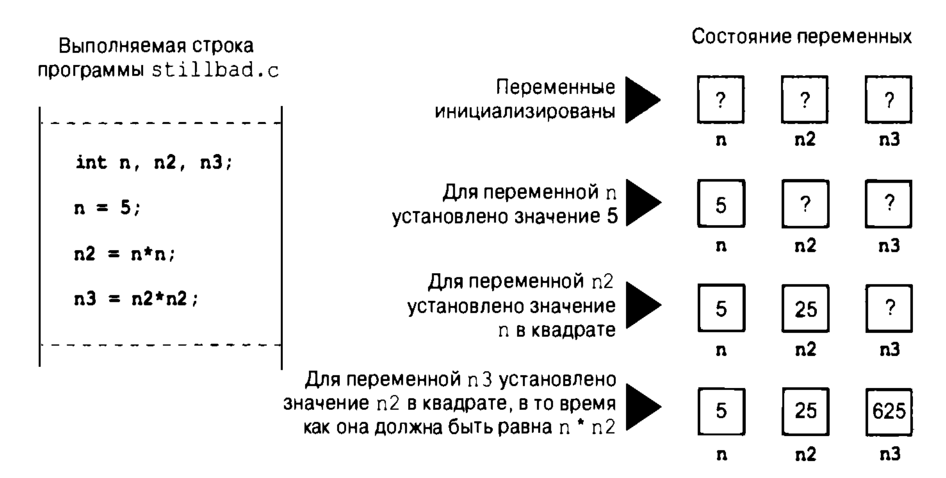


Рис. 2.6. Трассировка программы

Состояние программы — это просто набор значений всех переменных в заданной точке ее выполнения. Другими словами, это моментальный снимок текущего состояния вычислений.

Мы обсудили только один метод отслеживания состояния: самостоятельное пошаговое выполнение программы. В программе, которая делает, скажем, 10 000 итераций, вы не справитесь с такой задачей. Но можно выполнить несколько итераций, чтобы узнать, делает ли программа то, что от нее ожидается. Тем не менее, всегда существует возможность, что вы выполните эти шаги, как задумали, а не так, как действительно реализовали их в программе, поэтому старайтесь неукоснительно придерживаться фактического кода.

Еще один подход к выявлению семантических ошибок заключается в помещении в разные места кода дополнительных операторов `printf()` для текущего контроля избранных переменных в ключевых точках программы. Наблюдение за тем, как меняются эти значения, возможно, подскажет вам, что происходит. После того, как вы добьетесь от программы удовлетворительной работы, можете убрать дополнительные операторы и выполнить повторную компиляцию.

Третий метод исследования состояний программы предусматривает применение отладчика. *Отладчик* — это программа, которая позволяет выполнять другую программу в пошаговом режиме и просматривать значения переменных в этой программе. Отладчики характеризуются различными уровнями удобства использования и сложности. Наиболее совершенные отладчики отображают строку исходного кода, выполняемую в текущий момент. Это особенно удобно при отладке программ с альтернативными путями выполнения, поскольку легко видеть, по какому конкретному пути продвигается выполнение. Если ваш компилятор снабжен отладчиком, уделите время на его изучение. Например, опробуйте его на программе, приведенной в листинге 2.4.

Ключевые слова и зарезервированные идентификаторы

Ключевые слова образуют словарь языка С. Поскольку они играют в С особую роль, их нельзя применять, например, в качестве идентификаторов либо имен переменных. Многие из этих ключевых слов описывают разнообразные типы данных, скажем, `int`.

Другие, такие как `if`, служат для управления порядком выполнения операторов программы. В приведенном ниже перечне ключевых слов языка полужирным выделены ключевые слова, добавленные стандартом C90, курсивом показаны ключевые слова, введенные стандартом C99, а полужирным курсивом — появившиеся в стандарте C11.

Ключевые слова ISO C

auto	extern	short	while
break	float	**signed**	***_Alignas***
case	for	sizeof	***_Alignof***
char	goto	static	***_Bool***
const	if	struct	***_Complex***
continue	*inline*	switch	***_Generic***
default	int	typedef	***_Imaginary***
do	long	union	***_Noreturn***
double	register	unsigned	***_Static_assert***
else	restrict	void	***#_Thread_local***
enum	return	**volatile**	

Если вы попытаетесь использовать ключевое слово, к примеру, для имени переменной, компилятор воспримет это как синтаксическую ошибку. Существуют и другие идентификаторы, называемые *зарезервированными идентификаторами*, которые также не должны применяться для этих целей. Они не приводят к возникновению синтаксических ошибок, поскольку являются допустимыми именами. Однако они уже задействованы в языке или право на их применение зарезервировано за самим языком, поэтому если вы начнете их использовать для каких-то других целей, то могут возникнуть проблемы. Зарезервированными идентификаторами являются идентификаторы, начинающиеся с символа подчеркивания, а также имена стандартных библиотечных функций наподобие `printf()`.

Ключевые понятия

Программирование представляет собой довольно трудное занятие. Оно требует абстрактного, концептуального мышления и одновременно пристального внимания к деталям. Вы обнаружите, что компиляторы требуют внимательного отношения к деталям. При разговоре с хорошо знакомыми людьми вы можете употребить некоторые слова не по правилам, допустить несколько грамматических ошибок, возможно, оставить какие-то предложения неоконченными, но ваши знакомые все равно поймут, что вы хотели выразить. Тем не менее, компилятор не допускает подобных вольностей; он придерживается принципа "почти правильно — это по-прежнему неправильно".

Компилятор ничем не поможет в концептуальных вопросах, поэтому в данной книге мы стремимся заполнить этот пробел, выделяя основные понятия в каждой главе.

В этой главе основной целью должно быть понимание того, что собой представляет программа на языке C. Программу можно считать подготовленным вами описанием желаемого поведения компьютера. Компилятор выполняет по-настоящему кропотливую работу по преобразованию такого описания в базовый машинный язык. (Для того чтобы вы оценили, насколько огромную работу делает компилятор, отметим, что он создает исполняемый файл размером 60 Кбайт из исходного файла размером всего 1 Кбайт; для представления даже простой программы на C требуется большой объем кода на машинном языке.) Поскольку истинный интеллект у компилятора отсутствует, вы должны представить свое описание в понятных ему терминах, и эти термины

являются формальными правилами, установленными стандартом языка С. (Несмотря на ограничивающий характер, это все же лучше, чем необходимость выражать такое описание непосредственно на машинном языке!)

Компилятор ожидает получения инструкций в специальном формате, который был подробно рассмотрен в данной главе. Ваша задача как программиста — выразить свои идеи по поводу того, каким образом программа должна себя вести внутри инфраструктуры, который компилятор, руководящийся стандартом С, сможет успешно обработать.

Резюме

Программа на языке С состоит из одной или большего числа функций С. Каждая программа на С должна содержать функцию по имени `main()`, поскольку именно эта функция вызывается при запуске программы. Простая функция состоит из заголовка, за которым следует открывающая фигурная скобка, далее идут операторы, образующие тело функции, а за ними — завершающая, или *закрывающая*, фигурная скобка.

Каждый оператор языка С является инструкцией компьютеру и обязательно заканчивается точкой с запятой. Оператор объявления назначает переменной имя и определяет тип данных, которые будет храниться в этой переменной. Имя переменной может служить примером идентификатора. Оператор присваивания устанавливает значение переменной или, пользуясь более общим термином, области хранения. Оператор вызова функции запускает на выполнение функцию с указанным именем. Когда вызванная функция завершается, управление в программе переходит к оператору, следующему за вызовом функции.

Функция `printf()` может применяться для вывода фраз и значений переменных.

Синтаксис языка — это набор правил, которые регламентируют способ объединения вместе допустимых операторов в этом языке. *Семантика* оператора представляет его смысловое значение. Компилятор помогает обнаруживать синтаксические ошибки, однако семантические ошибки в поведении программы проявляются только после того, как она будет скомпилирована. Выявление семантических ошибок может предусматривать мониторинг состояния программы, т.е. значений всех переменных, на каждом шаге выполнения программы.

И, наконец, *ключевые слова* образуют словарь языка С.

Вопросы для самоконтроля

Ответы на эти вопросы находятся в приложении А.

1. Как называются базовые модули программы на языке С?
2. Что такое синтаксическая ошибка? Приведите примеры синтаксической ошибки в контексте своего родного языка и языка С.
3. Что такое семантическая ошибка? Приведите примеры в контексте своего родного языка и языка С.
4. Джо из Индианы написал и представил вам на утверждение следующую программу. Помогите ему исправить ошибки.

```
include studio.h
int main{void} /* программа выводит количество недель в году /*
(
int s
s := 56;
print(В году s недель.);
return 0;
```

5. Предположим, что каждый из приведенных ниже примеров является частью завершенной программы. Что выведет каждая такая часть?

а.
```
printf("Бе, бе, Черная Овечка.");
printf("У тебя найдется шерсть для меня?\n");
```

б.
```
printf("Прочь!\nВот наглая свинья!\n");
```

в.
```
printf("Что?\nНе/пклюет?\n");
```

г.
```
int num;

num = 2;
printf("%d + %d = %d", num, num, num + num);
```

6. Какие из следующих слов являются ключевыми в C? `main`, `int`, `function`, `char`, `=`

7. Как вывести значения переменных `words` и `lines`, чтобы они отобразились в следующей форме:

```
Текст содержал 3020 слов и 350 строк.
```

Здесь 3020 и 350 представляют значения этих двух переменных.

8. Рассмотрим следующую программу:

```
#include <stdio.h>
int main(void)
{
   int x, y;

   x = 10;
   b = 5;      /* строка 7 */
   y = x + y;  /* строка 8 */
   x = x*y;    /* строка 9 */
   printf("%d %d\n", x, y);
   return 0;
}
```

Каким будет состояние программы после выполнения строки 7? Строки 8? Строки 9?

9. Взгляните на следующую программу:

```
#include <stdio.h>
int main(void)
{
int x, y;
x = 10;
y = 5;          /* строка 7 */
y = x + y;      /* строка 8 */
x = x*y;        /* строка 9 */
printf("%d %d\n", x, y);
return 0;
}
```

Каким будет состояние программы после выполнения строки 7? Строки 8? Строки 9?

Упражнения по программированию

Для изучения языка C одного лишь чтения книг недостаточно. Вы должны попробовать написать несколько простых программ, чтобы посмотреть, так ли все гладко, как это выглядело в главе.

Мы предоставим вам некоторые соображения, однако вы сами должны продумать решение существующих задач. Ответы на избранные упражнения по программированию вы найдете на веб-сайте издательства этой книги.

1. Напишите программу, которая использует первый вызов функции `printf()` для вывода своего имени и фамилии в одной строке, второй вызов `printf()`, чтобы вывести имя и фамилию в двух строках, и еще два вызова `printf()` для вывода имени и фамилии в одной строке. Выходные данные должны иметь следующий вид (но с указанием ваших персональных данных):

```
Иван Иванов     ←Первый оператор вывода
Иван            ←Второй оператор вывода
Иванов          ←По-прежнему второй оператор вывода
Иван Иванов     ←Третий и четвертый операторы вывода
```

2. Напишите программу, выводящую ваше имя и адрес.

3. Напишите программу, которая преобразует ваш возраст в полных годах в количество дней и отображает на экране оба значения. Не обращайте внимания на високосные годы.

4. Напишите программу, которая производит следующий вывод:

```
Он веселый молодец!
Он веселый молодец!
Он веселый молодец!
Никто не может это отрицать!
```

Вдобавок к функции `main()` в программе должны использоваться две определенные пользователем функции: `jolly()`, которая выводит сообщение "Он веселый молодец!" один раз, и `deny()`, выводящая сообщение в последней строке.

5. Напишите программу, которая производит следующий вывод:

```
Бразилия, Россия, Индия, Китай
Индия, Китай,
Бразилия, Россия
```

Вдобавок к функции `main()` в программе должны использоваться две определенные пользователем функции: `br()`, выводящую строку "Бразилия, Россия" один раз, и `ic()`, которая один раз выводит строку "Индия, Китай". Функция `main()` должна позаботиться о любых дополнительных задачах вывода.

6. Напишите программу, которая создает целочисленную переменную по имени `toes`. Программа должна присвоить переменной `toes` значение `10`. Наряду с этим, программа должна вычислить удвоенное значение `toes` и квадрат `toes`. Программа должна вывести все три значения, снабдив их соответствующими пояснениями.

7. Многие исследования показывают, что улыбка способствует успеху. Напишите программу, которая производит следующий вывод:

```
Улыбайся!Улыбайся!Улыбайся!
Улыбайся!Улыбайся!
Улыбайся!
```

В программе должна быть определена функция, которая отображает строку "Улыбайся!" один раз. Эта функция должна вызываться столько раз, сколько необходимо.

8. В языке С одна функция может вызывать другую. Напишите программу, которая вызывает функцию по имени `one_three()`. Эта функция должна вывести слово "один" в одной строке, вызвать функцию `two()`, а затем вывести слово "три" тоже в одной строке. Функция `two()` должна отобразить слово "два" в одной строке. Функция `main()` должна вывести слово "начинаем:" перед вызовом функции `one_three()` и слово "порядок!" после ее вызова. Таким образом, выходные данные должны иметь следующий вид:

```
начинаем:
один
два
три
порядок!
```

3

Данные в языке C

В ЭТОЙ ГЛАВЕ...

- Ключевые слова:
- int, short, long, unsigned, char, float, double, _Bool, _Complex, _Imaginary
- Операция:
- sizeof
- Функция:
- scanf()
- Базовые типы данных в языке C
- Различия между целочисленными данными и данными с плавающей запятой
- Написание констант и объявление переменных известных типов
- Использование функций printf() и scanf() для чтения и записи значений различных типов

Программы работают с данными. Вы вводите числа, буквы и слова в компьютер и ожидаете, что он выполнит над этими данными какие-то действия. Например, вам может потребоваться, чтобы компьютер рассчитал прибыль и отобразил на экране отсортированный список виноторговцев. В этой главе вы будете не просто читать о данных, но практически манипулировать ими, что намного интереснее.

В настоящей главе рассматриваются два больших семейства типов данных: целые числа и числа с плавающей запятой (или плавающей точкой, что является синонимом). В языке C имеется несколько разновидностей этих типов. Здесь вы узнаете, что собой представляют такие типы, как их объявлять и каким образом и когда их применять. Кроме того, вы поймете отличия между константами и переменными, а в качестве полезного дополнения напишете свою первую интерактивную программу.

Демонстрационная программа

И снова мы начнем с написания демонстрационной программы. Как и прежде, вы столкнетесь с несколькими новыми и незнакомыми деталями, которые мы вскоре проясним. Общий замысел программы должен быть очевиден, поэтому попробуйте скомпилировать и выполнить исходный код, показанный в листинге 3.1. В целях экономии времени можете не вводить комментарии.

Листинг 3.1. Программа platinum.c

```
/* platinum.c -- ваш вес в платиновом эквиваленте */
#include <stdio.h>
int main(void)
{
    float weight;   /* вес пользователя */
    float value;    /* платиновый эквивалент */

    printf("Хотите узнать свой вес в платиновом эквиваленте?\n");
    printf("Давайте подсчитаем.\n");
    printf("Пожалуйста, введите свой вес, выраженный в фунтах: ");

    /* получить входные данные от пользователя */
    scanf("%f", &weight);
    /* считаем, что цена родия равна $1700 за тройскую унцию */
    /* 14.5833 коэффициент для перевода веса, выраженного в фунтах, в тройские унции */
    value = 1700.0 * weight * 14.5833;
    printf("Ваш вес в платиновом эквиваленте составляет $%.2f.\n", value);
    printf("Вы легко можете стать достойным этого! Если цена платины падает,\n");
    printf("ешьте больше для поддержания своей стоимости.\n");

    return 0;
}
```

СОВЕТ. Сообщения об ошибках и предупреждения

Если вы введете код программы некорректно, скажем, пропустив точку с запятой, компилятор выдаст сообщение о синтаксической ошибке. Однако даже при правильном вводе программы компилятор может выдать предупреждение, подобное следующему: "Преобразование из `double` в `float` может привести к потере данных". Сообщение об ошибке означает, что вы сделали что-то неправильно, и программа компилироваться не будет. С другой стороны, *предупреждение* означает, что введенный код является допустимым, но может привести не к тому результату, который ожидался. Предупреждение не вызывает прекращение компиляции. Это конкретное предупреждение связано с тем, как в языке C обрабатываются числа, подобные `1700.0`. В данном примере это не проблема, и позже в главе будет объяснен смысл такого предупреждения.

При вводе этой программы, возможно, потребуется заменить число 1700.0 текущей ценой платины. Однако не следует каким-либо образом изменять значение 14.5833, представляющее число тройских унций в одном фунте. (В качестве меры веса для драгоценных металлов используются тройские унции; для измерения веса всего остального применяются фунты.)

Обратите внимание, что ввод веса означает набор на клавиатуре числа, представляющего значение веса, и затем нажатие клавиши <Enter> или <Return>. Нажатие клавиши <Enter> информирует компьютер о завершении ввода. В программе предполагается, что для указания веса будет введено некоторое число, например, 156, а не слова вроде очень большой. Ввод букв вместо цифр вызывает проблемы, для устранения которых должен применяться оператор if (описанный в главе 7), поэтому вводите подходящее число. Ниже приведен пример вывода программы:

```
Хотите узнать свой вес в платиновом эквиваленте?
Давайте подсчитаем.
Пожалуйста, введите свой вес, выраженный в фунтах: 156
Ваш вес в платиновом эквиваленте составляет $3867491.25.
Вы легко можете стать достойным этого! Если цена платины падает,
ешьте больше для поддержания своей стоимости.
```

Настройка программы

Если вывод программы быстро мелькает на экране, а затем исчезает даже после добавления строки getchar();, как было описано в главе 2, вызов этой функции нужно использовать дважды:

```
getchar();
getchar();
```

Функция getchar() считывает следующий введенный символ, поэтому программа вынуждена дожидаться ввода. В данном случае мы предоставили ввод, набрав число 156 и затем нажав клавишу <Enter> (или <Return>), что приводит к передаче символа новой строки. Таким образом, функция scanf() считывает число, первая функция getchar() считывает символ новой строки, а вторая функция getchar() вынуждает программу приостановить выполнение, дожидаясь дальнейшего ввода.

Что нового в этой программе?

В этой программе появилось несколько новых элементов языка С.

- Обратите внимание, что в программе используется новый вид объявления переменных. В предыдущих примерах применялся только целочисленный тип переменных (int), а здесь добавился тип с плавающей запятой (float), что позволяет поддерживать более широкий спектр данных. Тип float может хранить числа с плавающей запятой.
- В программе демонстрируются новые способы записи констант. Теперь в роли констант выступают числа с десятичной точкой.
- Для вывода значения переменной нового типа в функции printf() должен использоваться спецификатор %f. Модификатор .2 в спецификаторе %f служит для настройки внешнего вида вывода, так что после десятичной точки будут отображаться два знака.
- Для ввода данных в программу с клавиатуры применяется функция scanf(). Спецификатор %f в scanf() означает, что с клавиатуры должно считываться число с плавающей запятой, а &weight — что введенное число будет присвоено переменной по имени weight. В функции scanf() используется амперсанд (&) для указания на то, где можно найти переменную weight. В следующей главе это рассматривается более подробно, а пока просто поверьте, что он здесь необходим.

- Вероятно, главной новой характеристикой этой программы является то, что она интерактивна. Компьютер запрашивает у вас информацию и затем задействует введенное вами число. Работать с интерактивными программами намного интереснее, чем с их неинтерактивными разновидностями. Но важнее то, что интерактивный подход делает программы более гибкими. Например, показанная выше демонстрационная программа может применяться для пересчета любого разумного веса, а не только 156 фунтов. Такую программу не придется переписывать каждый раз, когда она потребуется новому пользователю. Эта интерактивность обеспечивается функциями `scanf()` и `printf()`. Функция `scanf()` читает данные с клавиатуры и делает их доступными в программе, а функция `printf()` принимает данные от программы и выводит их на экран. Вместе эти две функции позволяют установить двухсторонний обмен данными с компьютером (рис. 3.1), что делает работу с компьютером гораздо более увлекательной.

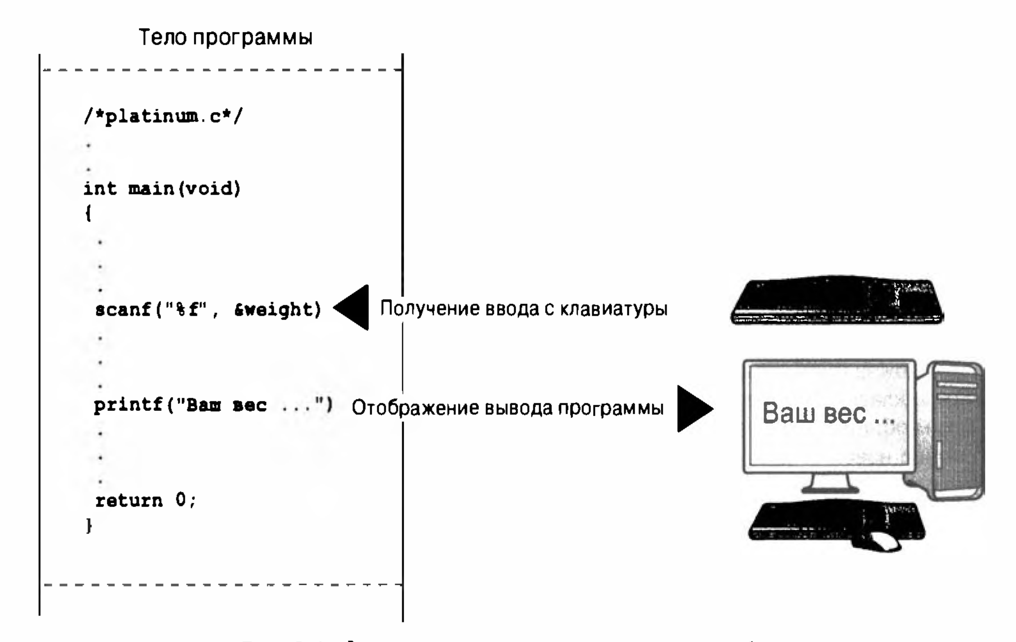


Рис. 3.1. Функции `scanf()` *и* `printf()` *в работе*

В настоящей главе рассматриваются два элемента из приведенного выше списка новых характеристик программы: переменные и константы различных типов данных. Оставшиеся три элемента исследуются в главе 4, но здесь мы продолжим в ограниченных масштабах пользоваться функциями `scanf()` и `printf()`.

Переменные и константы

Под руководством программы компьютер может выполнять множество действий. Он может суммировать числа, сортировать имена, воспроизводить аудио- и видеоклипы, вычислять орбиты комет, составлять списки адресатов почтовых отправлений, набирать телефонные номера, рисовать картинки, делать логические выводы и решать множество других задач, какие только можно себе вообразить. Для их решения программа должна работать с *данными*, т.е. числами и символами, несущими в себе необходимую информацию. Некоторые виды данных устанавливаются до начала выпол-

нения программы и их значения сохраняются неизменными в течение всего времени ее работы. Такие данные называются *константами*. Другие виды данных могут изменяться в ходе выполнения программы. Они называются *переменными*. В приведенной выше демонстрационной программе `weight` является переменной, а `14.5833` – константой. А что можно сказать о числе `1770.0`? Конечно, в реальности цена на платину не является постоянной величиной, но в этой программе она считается константой. Различие между переменной и константой состоит в том, что переменной можно присваивать значение либо изменять его во время выполнения, а с константой так поступать нельзя.

Ключевые слова для типов данных

Помимо отличий между переменными и константами, существует также разница между разных *типами* данных. Одни данные являются числами. Другие данные представляют собой буквы или, в общем случае, символы. Компьютеру необходим способ идентификации и использования этих разных видов данных. В языке C для этого предусмотрено несколько базовых *типов данных*. Если данные представляют собой константы, то обычно компилятор может выяснить их тип по внешнему виду: `42` – это целое число, а `42.100` – число с плавающей запятой. Тем не менее, тип переменной должен быть указан в операторе объявления. Позже вы узнаете, как объявлять переменные, но сначала давайте рассмотрим ключевые слова для базовых типов данных, распознаваемые языком C. В стандарте K&R C существовало семь ключевых слов, относящихся к типам. В стандарте C90 к этому списку были добавлены два ключевых слова. В стандарте C99 список пополнился еще тремя ключевыми словами (табл. 3.1).

Таблица 3.1. Ключевые слова для типов данных в языке C

Ключевые слова в исходном стандарте K&R C	Ключевые слова, добавленные стандартом C90	Ключевые слова, добавленные стандартом C99
`int`	`signed`	`_Bool`
`long`	`void`	`_Complex`
`short`		`_Imaginary`
`unsigned`		
`char`		
`float`		
`double`		

Ключевым словом `int` обозначается основной класс целых чисел, применяемых в C. Следующие три ключевых слова (`long`, `short` и `unsigned`) и добавленное стандартом C90 ключевое слово `signed` используются для указания вариаций этого базового типа, например, `unsigned short int` и `long long int`. С помощью ключевого слова `char` определяются символьные данные, к которым относятся буквы алфавита и другие символы, такие как `#`, `$`, `%` и `*`. Тип данных `char` можно также применять для представления небольших целых чисел. Типы `float`, `double` и `long double` служат для представления чисел с плавающей запятой. Тип данных `_Bool` используется для булевских значений (`true` и `false`), а типы данных `_Complex` и `_Imaginary` представляют, соответственно, комплексные и мнимые числа.

Типы данных, создаваемые с помощью указанных ключевых слов, могут быть разделены на два семейства на основе того, как они хранятся в памяти компьютера: *целочисленные* типы и типы *с плавающей запятой*.

Биты, байты и слова

Для описания элементов компьютерных данных или элементов компьютерной памяти могут применяться термины *бит*, *байт* и *слово*. Второму термину уделяется основное внимание.

Минимальная единица памяти называется *битом*, который может хранить одно из двух значений: `0` или `1`. (Иногда говорят, что бит "включен" или "выключен".) Конечно, в одном бите много информации сохранить не получится, но в компьютере их имеется огромное количество. Бит является базовым строительным блоком для памяти компьютера.

Байт — это наиболее часто используемая единица памяти компьютера. Практически на всех машинах байт состоит из 8 битов, и это является стандартным определением байта, по крайней мере, когда речь идет об измерении объема памяти. (Однако, как будет показано в разделе "Использование символов: тип `char`" далее в главе, в языке С имеется другое определение.) Поскольку бит может принимать значение 0 или 1, байт обеспечивает 256 (т.е. 2^8) возможных комбинаций нулей и единиц. Эти комбинации могут использоваться, например, для представления целых чисел от 0 до 255 или набора символов. Числа можно записывать посредством двоичного кода, в котором для представления чисел применяются только нули и единицы. (Двоичный код подробно рассматривается в главе 15, и при желании можете ознакомиться с начальными сведениями из указанной главы прямо сейчас.)

Слово — это естественная единица памяти для компьютера конкретного типа. В 8-разрядных микрокомпьютерах, таких как первые машины Apple, слово состояло из 8 битов. С тех пор персональные компьютеры перешли на 16-битные, 32-битные, а в настоящее время и 64-битные слова. Большие размеры слова позволяют быстрее передавать данные и делают доступным больший объем памяти.

Сравнение целочисленных типов и типов с плавающей запятой

Целочисленные типы? Типы с плавающей запятой? Если эти понятия выглядят совершенно незнакомыми, не переживайте. Вскоре будут предоставлены краткие пояснения. Если же вы не знаете, что такое биты, байты и слова, то первым делом прочитайте приведенную выше врезку. Должны ли вы изучить все до мельчайших деталей? Не обязательно. Вы ведь не обязаны знать все принципы работы двигателя внутреннего сгорания лишь для того, чтобы водить автомобиль; однако наличие некоторого представления о том, что именно происходит внутри компьютера или двигателя, иногда может помочь.

Для человека различие между целыми числами и числами с плавающей запятой отражено в способе их написания. Для компьютера это различие проявляется в способе, которым они хранятся в памяти. Давайте взглянем по очереди на оба класса данных.

Целые числа

Целое число — это число без дробной части. В языке С целое число никогда не записывается с десятичной точкой, например, 2, -23 и 2456. Числа наподобие 3.14, 0.22 и 2.000 целыми не являются. Целые числа хранятся в двоичной форме. Например, целое число 7 записывается в двоичной форме как 111. Следовательно, чтобы сохранить это число в 8-битном байте, нужно просто установить первые 5 битов в 0, а последние три бита — в 1 (рис. 3.2).

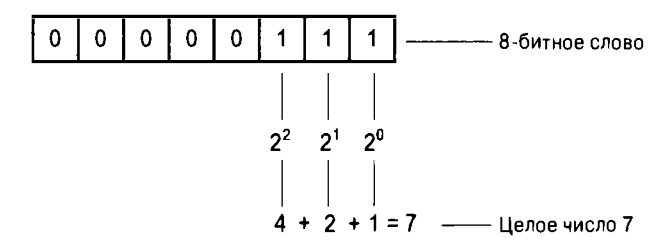


Рис. 3.2. Хранение целого числа 7 в двоичном коде

Числа с плавающей запятой

Число *с плавающей запятой* более или менее соответствует тому, что математики называют *вещественным числом.* К вещественным числам относятся числа, находящиеся в промежутках между целыми числами. Примерами чисел с плавающей запятой могут служить 2.75, 3.16E7, 7.00 и 2e-8. Обратите внимание, что добавление десятичной точки превращает целое число в число с плавающей запятой. Таким образом, 7 имеет целочисленный тип, но 7.00 – тип с плавающей запятой. Очевидно, что существует более одной формы записи числа с плавающей запятой. Более подробно экспоненциальную форму записи чисел мы обсудим позже, но если кратко, то запись 3.16E7 означает, что число 3.16 необходимо умножить на 10^7, т.е. на число, состоящее из единицы с последующими семью нулями. Число 7 называется *порядком* числа 10.

Ключевым моментом здесь является то, что схема, используемая для хранения числа с плавающей запятой, отличается от схемы, которая применяется для хранения целого числа. Число с плавающей запятой разделяется на дробную часть и порядок, которые хранятся отдельно. Таким образом, 7.00 будет храниться в памяти не в том виде, в каком хранится целое число 7, хотя оба они имеют одно и то же значение. Десятичным аналогом записи 7.00 может быть 0.7E1. Здесь 0.7 – дробная часть числа, а 1 – порядок. На рис. 3.3 показан еще один пример хранения числа с плавающей запятой. Разумеется, для хранения компьютер использует двоичные числа и степени 2, а не степени 10. Дополнительный материал по этой теме вы найдете в главе 15. А теперь сосредоточим внимание на практических различиях.

- Целое число не имеет дробной части; число с плавающей запятой может иметь дробную часть.
- Диапазон допустимых значений у чисел с плавающей запятой гораздо шире диапазона допустимых значений у целых чисел. Загляните в табл. 3.3, приведенную ближе к концу главы.
- При выполнении некоторых арифметических операций с плавающей запятой, таких как вычитание одного большого числа из другого, возможна значительная потеря точности.
- Поскольку в любом диапазоне чисел имеется бесконечное количество вещественных чисел, например, в диапазоне между 1.0 и 2.0, применяемые в компьютере числа с плавающей запятой не могут представить все числа этого диапазона. Числа с плавающей запятой часто являются приближениями настоящих значений. Например, 7.0 может быть сохранено как значение с плавающей запятой 6.99999 (вопросы точности более подробно рассматриваются позже).
- Раньше операции над числами с плавающей запятой выполнялись значительно медленнее операций над целыми числами. Однако многие современные ЦП содержат процессоры с плавающей запятой, которые сводят на нет эту проблему.

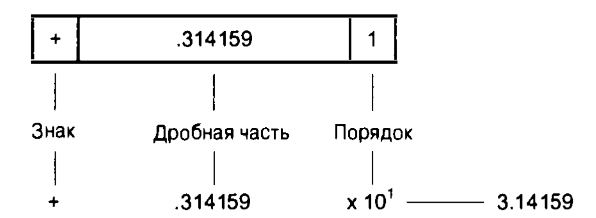


Рис. 3.3. Хранение числа π в формате числа с плавающей запятой (десятичная версия)

Базовые типы данных языка C

Давайте взглянем на особенности базовых типов данных, используемых в языке C. Для каждого типа мы покажем, как объявлять переменные и представлять константы с фиксированными значениями наподобие 5 или 2.78, а также продемонстрируем типичные случаи их применения. Некоторые старые компиляторы C поддерживают не все эти типы данных, поэтому выясните в документации, какие типы данных доступны в компиляторе.

Тип int

Язык C предлагает множество целочисленных типов, и вы, скорее всего, хотите знать, почему одного типа оказалось не достаточно. Дело в том, что язык C предоставляет программисту возможность сопоставления типа с конкретным случаем использования. В частности, целочисленные типы C варьируются в диапазонах допустимых значений и в возможности применения отрицательных чисел. Тип int является базовым выбором, но если вам потребуются другие варианты, удовлетворяющие требованиям определенной задачи или компьютера, то они также доступны.

Тип int представляет целое число со знаком. Это значит, что оно должно быть целым и может иметь положительную, отрицательную или нулевую величину. Диапазон возможных значений зависит от компьютерной системы. Обычно для хранения данных типа int используется одно машинное слово. Поэтому в компьютерах, совместимых со старыми моделями IBM PC с 16-битными словами, для хранения данных типа int выделялось 16 битов. Это позволяло иметь диапазон значений от -32768 до 32767. Современные персональные компьютеры обычно оперируют 32-битными целыми числами и данные типа int соответствуют такому размеру. В настоящее время индустрия персональных компьютеров сориентировалась на выпуск 64-разрядных процессоров, которые могут свободно манипулировать еще большими целыми числами. В стандарте ISO C указано, что минимальным диапазоном для типа int должен быть от -32767 до 32767. Обычно системы представляют целые числа со знаком за счет использования значения определенного бита. Распространенные способы представления рассматриваются в главе 15.

Объявление переменной типа int

Как было показано в главе 2, для объявления целочисленных переменных применяется ключевое слово int. Сначала указывается ключевое слово int, затем выбранное имя для переменной и, наконец, точка с запятой. Объявление более одной переменной можно делать либо по отдельности, либо поместить после ключевого слова int список имен, отделяя их друг от друга запятыми. Ниже показаны примеры допустимых объявлений:

```
int erns;
int hogs, cows, goats;
```

Для каждой переменной можно было бы предусмотреть отдельное объявление или же объявить все четыре переменных в одном операторе. Результат будет таким же: связывание имен с выделенными областями памяти для четырех переменных типа int.

Эти объявления создают переменные, но не присваивают им значения. Каким образом переменные получают значения? Вы уже видели два способа, с помощью которых переменные могут получать значения в программе. Первый способ — оператор присваивания:

```
cows = 112;
```

Второй способ предусматривает получение переменной значения из функции, например, из scanf(). А теперь рассмотрим третий способ.

Инициализация переменных

Инициализация переменной означает присваивание ей *начального* значения. В языке С это можно делать в виде части объявления. Достаточно после имени переменной поместить операцию присваивания (=) и указать значение, которое переменная должна получить. Вот несколько примеров:

```
int hogs = 21;
int cows = 32, goats = 14;
int dogs, cats = 94;       /* допустимая, но неудачная форма */
```

В последней строке инициализируется только переменная cats. Однако по невнимательности может показаться, что переменная dogs также инициализируется значением 94, поэтому лучше избегать использования в одном операторе объявления инициализированных и неинициализированных переменных.

Выражаясь кратко, эти объявления выделяют и помечают для переменных области хранения, а также присваивают им начальные значения (рис. 3.4).

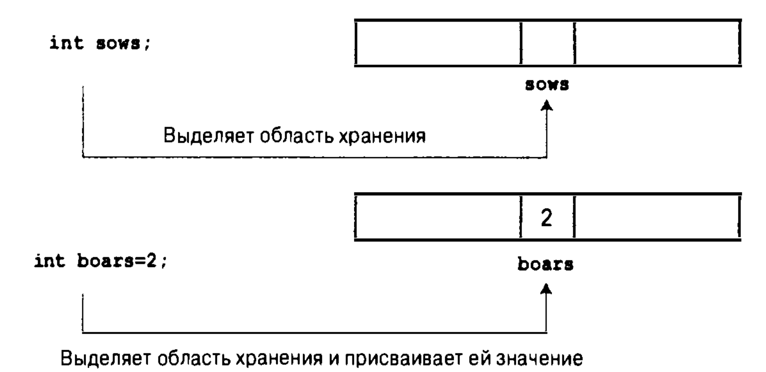


Рис. 3.4. Определение и инициализация переменной

Константы типа int

Различные целые числа (21, 32, 14 и 94) в последнем примере являются *целочисленными константами*, также называемыми *целочисленными литералами*. Когда вы записываете число без десятичной точки и порядка, компилятор С распознает его как целое. Следовательно, числа 22 и -44 являются целочисленными константами, а числа 22.0 и 2.2E1 — нет. Большинство целочисленных констант трактуются в С как принадлежащие к типу int. Очень большие целые числа могут трактоваться иначе; в разделе "Константы long и long long" далее в главе рассматриваются данные типа long int.

Вывод значений типа int

Для вывода значений типа int можно применять функцию printf(). Как уже упоминалось в главе 2, конструкция %d служит для указания в строке места, где будет выводиться целое число. Конструкция %d называется *спецификатором формата*, поскольку она определяет формат, используемый функцией printf() для отображения конкретного значения. Каждому спецификатору %d в строке формата должно соответствовать значение int в списке выводимых элементов. Таким значением может быть переменная int, константа int или любое другое выражение int. Программист должен следить за тем, чтобы количество спецификаторов формата соответствовало числу значений, потому что компилятор не обнаруживает ошибки подобного рода. В листинге 3.2 представлена простая программа, которая инициализирует переменную, а затем выводит значение этой переменной, значение константы и значение простого выражения. Вдобавок она демонстрирует, что происходит в случае невнимательности.

Листинг 3.2. Программа print1.c

```
/* print1.c -- демонстрирует некоторые свойства функции printf() */
#include <stdio.h>
int main(void)
{
    int ten = 10;
    int two = 2;

    printf("Выполняется правильно: ");
    printf("%d минус %d равно %d\n", ten, 2, ten - two );
    printf("Выполняется неправильно: ") ; ");
    printf("%d минус %d равно %d\n", ten );  // пропущены 2 аргумента

    return 0;
}
```

Компиляция и запуск этой программы ведет к получению следующего вывода:

```
Выполняется правильно: 10 минус 2 равно 8
Выполняется неправильно: 10 минус 16 равно 1650287143
```

В первой строке вывода первый спецификатор %d представляет переменную ten типа int, второй – константу 2 типа int и третий – значение выражения ten - two типа int. Однако во второй строке переменная ten применяется для предоставления значения только первому спецификатору %d, а для последующих двух спецификаторов %d значений не предусмотрено, поэтому программа использует для них случайные значения, находящиеся в памяти! (На своем компьютере вы можете получить результат, сильно отличающийся от полученного в этом примере. Может отличаться не только содержимое памяти, но также разные компиляторы будут управлять ячейками памяти по-разному.)

Вас может раздражать тот факт, что компилятор не выявляет настолько очевидную ошибку. Причина связана с необычным поведением функции printf(). Большинство функций принимают заданное количество аргументов, и компилятор может проверить правильность указанного числа аргументов. Однако функция printf() может принимать один, два, три и большее количество аргументов, и это препятствует применению компилятором обычных методов обнаружения ошибок такого рода. Однако некоторые компиляторы будут использовать нестандартные методы проверки, и предупреждать о возможных неправильных действиях. Тем не менее, лучше всего всегда проверять в функции printf() соответствие количества спецификаторов формата количеству значений, которые подлежат отображению на экране.

Восьмеричные и шестнадцатеричные числа

Обычно в языке C предполагается, что целочисленные константы являются десятичными числами (по основанию 10). Однако у многих программистов пользуются популярностью восьмеричные (по основанию 8) и шестнадцатеричные (по основанию 16) числа. Поскольку 8 и 16 представляют собой степени числа 2, а 10 — нет, восьмеричная и шестнадцатеричная системы счисления более удобны для представления чисел, связанных с компьютерами. Например, число 65536, которое часто встречается в 16-разрядных машинах, в шестнадцатеричной форме записывается как 10000. Кроме того, каждая цифра шестнадцатеричного числа соответствует в точности 4 битам. Например, шестнадцатеричная цифра 3 — это 0011, а шестнадцатеричная цифра 5 — это 0101. Таким образом, шестнадцатеричному значению 35 соответствует битовая комбинация 0011 0101, а шестнадцатеричному значению 53 — 0101 0011. Такое соответствие позволяет облегчить переход от шестнадцатеричного представления числа к двоичному представлению (по основанию 2) и обратно. Но каким образом компьютер может определить, в какой форме записано число 10000 — в десятичной, шестнадцатеричной или восьмеричной? В языке C система счисления задается с помощью специального префикса. Префикс `0x` или `0X` означает, что вы указываете шестнадцатеричное число, поэтому 16 в шестнадцатеричном виде записывается как `0x10` или `0X10`. Аналогично, префикс `0` (ноль) означает, что задается восьмеричное число. Например, в C десятичное число 16 в восьмеричном виде записывается как `020`. Более подробно эти альтернативные системы рассматриваются в главе 15. Вы должны понимать, что возможность применения разных систем счисления является лишь удобством для программистов. Это не влияет на способ хранения числа в памяти. Другими словами, вы можете написать `16`, `020` или `0x10`, и это число в каждом случае будет храниться в памяти одинаковым образом — в двоичном коде, используемом внутри компьютера.

Отображение восьмеричных и шестнадцатеричных чисел

Язык C позволяет не только записывать число в любой из трех систем счисления, но и отображать его во всех них. Чтобы вывести на экран целое число в восьмеричном, а не десятичном виде, вместо `%d` применяйте спецификатор `%o`. Для отображения целого числа в шестнадцатеричном виде служит спецификатор `%x`. Если вы хотите вывести на экран префиксы языка C, воспользуйтесь спецификаторами `%#o`, `%#x` и `%#X`, которые позволяют отображать префиксы `0`, `0x` и `0X`. В листинге 3.3 приведен небольшой пример. (Вспомните, что в некоторых интегрированных средах разработки может потребоваться вставить в программу оператор `getchar();`, чтобы предотвратить немедленное закрытие окна выполнения программы.)

Листинг 3.3. Программа bases.c

```
/* bases.c -- выводит число 100 в десятичной, восьмеричной и шестнадцатеричной форме */
#include <stdio.h>
int main(void)
{
   int x = 100;
   printf("десятичное = %d; восьмеричное = %o; шестнадцатеричное = %x\n", x, x, x);
   printf("десятичное = %d; восьмеричное = %#o; шестнадцатеричное = %#x\n", x, x, x);
   return 0;
}
```

После компиляции и выполнения этой программы получен следующий вывод:

```
десятичное = 100; восьмеричное = 144; шестнадцатеричное = 64
десятичное = 100; восьмеричное = 0144; шестнадцатеричное = 0x64
```

Одно и то же значение отображается в трех различных системах счисления. Все преобразования выполняет функция printf(). Обратите внимание, что префиксы 0 или 0x не отображаются в выводе до тех пор, пока в спецификаторе не будет указан символ #.

Другие целочисленные типы

Если вы просто изучаете язык С, то в большинстве случаев вам, скорее всего, вполне достаточно будет типа int. Однако для полноты картины мы рассмотрим и другие формы целых чисел. При желании вы можете пропустить этот раздел и перейти к обсуждению типа char в разделе "Использование символов: тип char", а затем возвратиться к данному разделу в случае необходимости.

В языке С применяются три ключевых слова, модифицирующих базовый целочисленный тип: short, long и unsigned. Примите во внимание следующие аспекты.

- Тип short int, или short, может использовать меньший объем памяти, чем int, и тем самым экономить память в случае, когда требуются только небольшие числа. Подобно int, short является типом со знаком.
- Тип long int, или long, может занимать больший объем памяти, чем int, позволяя представлять крупные целочисленные значения. Подобно int, long является типом со знаком.
- Тип long long int, или long long (введен стандартом C99), может занимать больше памяти, чем long. Для этого типа используются минимум 64 бита. Подобно int, long long является типом со знаком.
- Тип unsigned int, или unsigned, применяется для переменных, которые принимают только неотрицательные значения. Этот тип сдвигает диапазон хранимых чисел. Например, 16-битный тип unsigned int имеет диапазон значений от 0 до 65535 вместо диапазона от -32768 до 32767. Бит, который использовался для представления знака, теперь становится еще одной двоичной цифрой, делая возможным представление большего числа.
- Типы unsigned long int, или unsigned long, и unsigned short int, или unsigned short, распознаются как допустимые стандартом C90. В стандарте C99 к ним добавлен тип unsigned long long int, или unsigned long long.
- Ключевое слово signed может применяться с любыми типами со знаком, чтобы явно указать свое намерение. Например, short, short int, signed short и signed short int являются именами одного и того же типа.

Объявление переменных других целочисленных типов

Переменные других целочисленных типов объявляются в той же манере, что и переменные типа int. Ниже приведен список примеров объявлений. Не все старые компиляторы С распознают последние три объявления, а финальное объявление соответствует стандарту C99.

```
long int estine;
long johns;
short int erns;
short ribs;
unsigned int s_count;
unsigned players;
unsigned long headcount;
unsigned short yesvotes;
long long ago;
```

Почему доступно так много целочисленных типов?

Почему мы говорим, что типы long и short "могут" использовать больше или меньше памяти, чем int? Дело в том, что язык C гарантирует только то, что тип short не будет длиннее типа int, а тип long не будет короче типа int. Идея заключается в том, чтобы согласовать типы с конкретной машиной. Например, во времена операционной системы Windows 3 типы int и short были 16-битными, a long – 32-битным. Позже системы Windows и Apple перешли на использование 16 битов для типа short и 32 битов для типов int и long. Применение 32 битов расширяет диапазон допустимых целых чисел до более 2 миллиардов. В настоящее время, когда распространенными стали 64-разрядные процессоры, возникла потребность в 64-битных целых числах, что и стало причиной появления типа long long. Чаще всего в наши дни тип long long устанавливается как 64-битный, long – 32-битный, short – 16-битный, а int – либо 16-битный, либо 32-битный, в зависимости от естественного размера машинного слова. В принципе эти четыре типа могли бы представлять четыре разных размера, но на практике, по меньшей мере, некоторые из них перекрываются.

В стандарте языка C предоставлены указания по минимально допустимому размеру для каждого базового типа данных. Минимальный диапазон значений для типов short и int составляет от -32 767 до 32 767, соответствуя 16-битной единице памяти, а минимальный диапазон для типа long – от -2 147 483 647 до 2 147 483 647, что соответствует 32-битной единице. Для типов unsigned short и unsigned int минимальный диапазон охватывает числа от 0 до 65 535, а для типа unsigned long он находится в пределах от 0 до 4 294 967 295. Тип long long предназначен для поддержки 64-битных данных. Его минимальный диапазон довольно внушителен и простирается от -9 223 372 036 854 775 807 до 9 223 372 036 854 775 807. Минимальный диапазон для типа unsigned long long охватывает числа от 0 до 18 446 744 073 709 551 615.

Когда должны использоваться разнообразные вариации типа int? Для начала рассмотрим типы без знака. Их вполне естественно применять для подсчета, поскольку в таких случаях отрицательные числа не нужны, к тому же типы без знака делают возможными более высокие положительные значения, чем типы со знаком.

Тип long должен использоваться, когда необходимы числа, которые он позволяет поддерживать, а int – нет. Тем не менее, в системах, где тип long длиннее int, применение типа long может замедлить вычисления, поэтому его не стоит использовать без крайней необходимости. Еще один момент: если вы пишете код для машины, на которой типы int и long имеют один и тот же размер, а вам нужны 32-битные целые числа, то выбирайте тип long, а не int, чтобы программа функционировала корректно в случае переноса на 16-разрядную машину. Аналогично, применяйте тип long long, если требуются 64-битные целочисленные значения.

Пользуйтесь типом short в целях экономии памяти, например, в случае, когда нужно 16-битное значение в системе, в которой int занимает 32 бита. Обычно экономия памяти важна, только если в программе обрабатываются массивы целых чисел, которые велики по сравнению с доступной памятью системы. Еще одна причина для применения типа short связана с тем, что по своему размеру он может соответствовать аппаратным регистрам, используемым отдельными компонентами системы.

Целочисленное переполнение

Что произойдет, если целое число окажется больше, чем допускает выбранный для него тип? Давайте присвоим целочисленной переменной максимально возможное целое значение, добавим к нему еще какое-то целое число и посмотрим, к чему это приведет. Мы выполним это действие над типами со знаком и без знака. (В вызове функции printf() для отображения значений типа unsigned int применяется спецификатор %u.)

```
/* toobig.c -- превышение максимально возможного значения int в системе */
#include <stdio.h>
int main(void)
{
    int i = 2147483647;
    unsigned int j = 4294967295;

    printf("%d %d %d\n", i, i+1, i+2);
    printf("%u %u %u\n", j, j+1, j+2);

    return 0;
}
```

В нашей системе был получен следующий результат:

```
2147483647 -2147483648 -2147483647
4294967295 0 1
```

Целочисленная переменная без знака j действует как счетчик пробега автомобиля. Когда достигается максимальное значение, оно сбрасывается, и подсчет начинается с начала. Целочисленная переменная i ведет себя аналогично. Главное различие между ними заключается в том, что значения переменной j типа unsigned int, подобно счетчику пробега, начинаются с 0, в то время как значения переменной i типа int — с -2 147 483 648. Обратите внимание, что о превышении максимального значения (переполнении) переменной i ничего не сообщается. Чтобы отслеживать это, вам придется самостоятельно предусмотреть подходящий код.

Описанное здесь поведение программы регламентируется правилами языка С для беззнаковых типов. В стандарте не определено, как должны себя вести типы со знаком. Показанное здесь поведение является типовым, но вы вполне можете столкнуться с другим поведением.

Константы long и long long

Обычно, когда в коде программы используется число вроде 2345, оно хранится в памяти как относящееся к типу int. А что произойдет, если указать число, такое как 1 000 000, в системе, где тип int не способен хранить настолько большое значение? В этом случае компилятор трактует его как число типа long int, предполагая, что этого типа окажется достаточно. Если число превосходит максимально возможное значение типа long, оно будет рассматриваться как значение типа unsigned long. Если и этого не достаточно, оно интерпретируется как значение типа long long или unsigned long long, если данные типы доступны.

Восьмеричные и шестнадцатеричные константы трактуются как значения типа int, если их значение не слишком велико. Затем компилятор примеривает к ним тип unsigned int. Если и его не хватает, компилятор последовательно пробует типы long, unsigned long, long long и unsigned long long.

Иногда необходимо, чтобы компилятор сохранил небольшое число как целое значение типа long. Например, это может потребоваться при явном использовании в коде адресов памяти в IBM PC. Кроме того, некоторые стандартные функции С требуют значений типа long. Чтобы небольшая константа интерпретировалась как значение типа long, к ней можно дописать букву l (строчная буква L) или L. Вторая форма предпочтительнее, поскольку она не выглядит похожей на цифру 1. Следовательно, система с 16-битным типом int и 32-битным типом long трактует целое число 7 как 16-битное, а целое число 7L – как 32-битное. Суффиксы l и L можно также применять с восьмеричными и шестнадцатеричными числами, например, 020L и 0x10L. Аналогично, в системах, поддерживающих тип long long, можно использовать суффикс ll или LL для указания значения типа long long, например, 3LL. Чтобы задать тип unsigned long long, добавьте к суффиксу букву u или U, как в 5ull, 10LLU, 6LLU и 9Ull.

Вывод значений типов short, long, long long и unsigned

Для вывода чисел типа unsigned int применяйте спецификатор %u. Чтобы вывести значение типа long, используйте спецификатор формата %ld. Если типы int и long в вашей системе имеют один и тот же размер, вполне достаточно спецификатора %d, однако ваша программа не будет корректно работать при переносе в систему, где эти два типа обладают разными размерами, поэтому для long лучше применять спецификатор %ld. Вместе с префиксами x и o можно также указывать префикс l. Таким образом, вы можете использовать спецификатор %lx для вывода целого числа типа long в шестнадцатеричном формате и спецификатор %lo – для его вывода в восьмеричном формате. Обратите внимание, что хотя язык C позволяет применять в качестве суффиксов констант и прописные, и строчные буквы, в этих спецификаторах формата используются только строчные буквы.

В языке C доступны дополнительные форматы для printf(). Первым делом, можно применять префикс h для значений типа short. Следовательно, спецификатор %hd отображает целое число типа short в десятичной форме, а спецификатор %ho отображает это же число в восьмеричной форме. Префиксы h и l можно использовать вместе с префиксом u для типов без знака. Например, для вывода значений типов unsigned long можно было бы указать %lu. В листинге 3.4 приведен пример. В системах, поддерживающих типы long long, для версий со знаком и без знака применяются спецификаторы %lld и %llu. Более полное обсуждение спецификаторов формата можно найти в главе 4.

Листинг 3.4. Программа print2.c

```
/* print2.c -- дополнительные свойства функции printf() */
#include <stdio.h>
int main(void)
{
    unsigned int un = 3000000000;  /* система с 32-битным типом int */
    short end = 200;               /* и 16-битным типом short */
    long big = 65537;
    long long verybig = 12345678908642;

    printf("un = %u, но не %d\n", un, un);
    printf("end = %hd и %d\n", end, end);
    printf("big = %ld, но не %hd\n", big, big);
    printf("verybig = %lld, но не %ld\n", verybig, verybig);

    return 0;
}
```

Ниже показан вывод в конкретной системе (результаты могут варьироваться):

```
un = 3000000000, но не -1294967296
end = 200 и 200
big = 65537, но не 1
verybig = 12345678908642, но не 1942899938
```

Этот пример демонстрирует, что использование неправильных спецификаторов может привести к неожиданным результатам. Прежде всего, обратите внимание, что применение спецификатора %d для беззнаковой переменной un выдает отрицательное число! Причина в том, что значение 3 000 000 000 без знака и значение –129 496 296 со знаком имеют одно и то же внутреннее представление в памяти нашей системы. (В главе 15 это свойство объясняется более подробно.) Таким образом, если указать функции printf(), что значение является числом без знака, она выведет одно зна-

чение, а если указать, что значение представляет собой число со знаком, то другое значение. Подобное поведение происходит для значений, которые превышают максимально допустимое значение для типа со знаком. Небольшие положительные значения, такие как 96, сохраняются и отображаются одинаково как для типов со знаком, так и для типов без знака.

Далее отметим, что переменная end типа short отображается одинаково независимо от того, указываете вы функции printf() принадлежность end к типу short (спецификатор %hd) или к типу int (спецификатор %d). Это объясняется тем, что при передаче аргумента функции С значение типа short автоматически расширяется до типа int. Здесь могут возникнуть два вопроса: почему предпринимается указанное преобразование, и для чего используется модификатор h? Ответ на первый вопрос прост: для типа int выбирался такой размер, чтобы обеспечить наиболее эффективную его обработку компьютером. Следовательно, на компьютере, в котором типы short и int имеют разные размеры, передача значения как int может осуществляться быстрее. Ответ на второй вопрос выглядит так: модификатор h можно применять, чтобы продемонстрировать, какой вид примет целое значение, будучи усеченным до типа short. Иллюстрацией этого утверждения может служить третья строка вывода. Число 65537, записанное в двоичном формате как 32-битное число, имеет вид 00000000000000010000000000000001. С помощью спецификатора %hd мы заставляем функцию printf() просматривать только последние 16 битов числа, поэтому она отображает в результате 1. Аналогично, финальная строка вывода показывает полное значение verybig, после чего это значение сохраняется в последних 32 битах, на что указывает спецификатор %ld.

Как упоминалось ранее, именно программист отвечает за то, чтобы количество спецификаторов соответствовало количеству отображаемых значений. Теперь вы видите, что программист несет ответственность также и за правильный выбор спецификаторов, соответствующих типу отображаемых значений.

СОВЕТ. Соответствие типов и спецификаторов в printf()

Не забывайте проверять, что для каждого отображаемого значения в операторе printf() предусмотрен один спецификатор формата. Кроме того, проверяйте, что тип каждого спецификатора формата соответствует типу отображаемого значения.

Использование символов: тип char

Тип данных char применяется для хранения символов, таких как буквы и знаки препинания, однако формально он также является целочисленным. Почему? Причина в том, что тип char в действительности хранит целые числа, а не символы. Для поддержки символов компьютер использует числовой код, в котором определенные целые числа представляют определенные символы. В США наиболее часто применяется код ASCII (приложение В), и он как раз принят в настоящей книге. К примеру, целое значение 65 в нем представляет прописную букву *А*. Таким образом, чтобы сохранить букву *А*, фактически нужно записать целое число 65. (Во многих мэйнфреймах IBM используется другой код, EBCDIC, но принцип остается тем же. В компьютерных системах, эксплуатируемых в других странах, могут применяться совершенно другие коды.)

Стандартный код ASCII состоит из последовательности чисел от 0 до 127. Этот диапазон достаточно мал, чтобы для значения хватило 7 битов. Тип char обычно определяется как 8-битная единица памяти, поэтому ее более чем достаточно, чтобы уместить стандартный код ASCII. Во многих системах, таких как IBM PC и Apple Macintosh, используются расширенные коды ASCII (разные для этих двух систем), которые по-прежнему не выходят за пределы 8 битов. В общем случае язык С гаранти-

рует, что тип char достаточно велик, чтобы представлять базовый набор символов в системах, для которых реализованы компиляторы С.

Многие наборы символов содержат более 127 или даже 255 значений. Например, существует набор символов Japanese kanji для японских иероглифов. В рамках коммерческой инициативы Unicode был создана система для представления широкого разнообразия наборов символов, применяемых в различных частях мира, которая в настоящее время содержит более 110 000 символов. Организация ISO и комиссия IEC (International Electrotechnical Commission — Международная электротехническая комиссия) вместе разработали для наборов символов стандарт, получивший название ISO/IEC 10646. К счастью, стандарт Unicode сохранил совместимость с более широким стандартом ISO/IEC 10646.

Язык С определяет байт как несколько битов, используемых типом char, поэтому может быть система с 16- или 32-битным байтом и типом char.

Объявление переменных типа char

Как и можно было ожидать, переменные типа char объявляются в такой же манере, что и другие переменные. Вот несколько примеров:

```
char response;
char itable, latan;
```

В этом коде создаются три переменных типа char: response, itable и latan.

Символьные константы и инициализация

Предположим, что вы хотите инициализировать символьную константу буквой *А*. Компьютерные языки призваны облегчить решение этой задачи, так что вам не придется запоминать все коды ASCII. Вы можете присвоить символ A переменной grade с помощью следующей инициализации:

```
char grade = 'A';
```

Одиночный символ, заключенный в одиночные кавычки, представляет собой *символьную константу* в С. Когда компилятор встречает конструкцию 'A', он преобразует ее в подходящее кодовое значение. Одиночные кавычки здесь очень важны. Рассмотрим еще один пример:

```
char broiled;  /* объявление переменной типа char                          */
broiled = 'T'; /* правильно                                                 */
broiled = T;   /* Неправильно! Компилятор считает, что T является переменной  */
broiled = "T"; /* Неправильно! Компилятор считает, что "T" является строкой */
```

Если опустить кавычки, то компилятор посчитает, что T является именем переменной. Если применить двойные кавычки, он воспримет "T" как строку. Строки рассматриваются в главе 4.

Поскольку символы в действительности хранятся в виде числовых значений, для присваивания значений можно также указывать числовые коды:

```
char grade = 65; /* правильно в контексте ASCII, но стиль неудачен */
```

В данном примере 65 имеет тип int, но поскольку это значение меньше максимального значения типа char, оно может быть присвоено переменной grade без каких-либо проблем. Так как 65 представляет собой ASCII-код буквы *А*, в этом примере переменной grade присваивается значение A. Тем не менее, обратите внимание, что в примере предполагается использование в системе кодировки ASCII. Указание 'A' вместо 65 дает в результате код, который работает в любой системе. Таким образом, применять символьные константы намного лучше, чем значения числовых кодов.

Несколько странно, однако C трактует символьные константы как тип int, а не char. Например, в системе с 32-битным типом int и с 8-битным типом char следующий код представляет 'B' как числовое значение 66, хранящееся в 32-битной единице памяти, но переменная grade в итоге получает значение 66 в 8-битной единице памяти:

```
char grade = 'B';
```

Эта характеристика символьных констант делает возможным определение символьной константы вида 'FATE', с четырьмя отдельными 8-битными ASCII-кодами, хранящимися в 32-битной единице памяти. Тем не менее, попытка присвоить такую символьную константу переменной типа char приводит к тому, что используются только последние 8 битов, так что переменная получает значение 'E'.

Непечатаемые символы

Прием с одиночными кавычками хорош для символов, цифр и знаков препинания, однако если просмотреть таблицу кодов ASCII, в ней можно обнаружить также непечатаемые символы. Например, некоторые из них представляют собой такие действия, как возврат на одну позицию влево, переход на следующую строку или выдачу звукового сигнала терминалом либо встроенным динамиком. Как их можно представить? В языке C предлагаются три способа. Первый способ уже упоминался – применение ASCII-кода. Например, ASCII-кодом для символа звукового сигнала является 7, так что можно использовать следующий оператор:

```
char beep = 7;
```

Второй способ представления необычных символов в языке C предусматривает применение специальных последовательностей символов, которые называются *управляющими последовательностями*. Список управляющих последовательностей и их описание приведено в табл. 3.2.

Таблица 3.2. Управляющие последовательности

Последовательность	Описание
\a	Предупреждение (стандарт ANSI C)
\b	Возврат на одну позицию влево
\f	Перевод страницы
\n	Новая строка
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\\	Обратная косая черта (\)
\'	Одиночная кавычка (')
\"	Двойная кавычка (")
\?	Знак вопроса (?)
\0*oo*	Восьмеричное значение (*o* представляет восьмеричную цифру)
\x*hh*	Шестнадцатеричное значение (*h* представляет шестнадцатеричную цифру)

Управляющие последовательности, присваиваемые символьным переменным, должны быть заключены в одиночные кавычки. Например, можно записать такой оператор:

```
char nerf = '\n';
```

а затем вывести переменную `nerf`, что обеспечит перемещение на следующую строку на принтере или на экране монитора.

Теперь давайте более пристально взглянем, что делает каждая управляющая последовательность. Символ предупреждения (`\a`), введенный стандартом C90, вызывает появление звукового или визуального предупреждающего сигнала. Природа предупреждающего сигнала зависит от оборудования; чаще других используется звуковой сигнал. (В некоторых системах предупреждающий символ не оказывает никакого действия.) Стандарт С требует, чтобы предупреждающий сигнал не изменял активную позицию. Под *активной позицией* в стандарте понимается место в устройстве отображения (экран, телетайп, принтер и т.д.), в котором иначе появился бы следующий символ. Выражаясь кратко, активная позиция – это обобщение понятия экранного курсора, с которым вы наверняка хорошо знакомы. Применение предупреждающего символа в программе, выводящей на экран, должно вызвать звуковой сигнал без перемещения экранного курсора.

Управляющие последовательности `\b`, `\f`, `\n`, `\r`, `\t` и `\v` представляют собой обычные символы управления выходным устройством. Их проще всего описывать в терминах того, как они влияют на активную позицию. Символ возврата на одну позицию влево (`\b`) перемещает активную позицию назад на один символ текущей строки. Символ перевода страницы (`\f`) переносит активную позицию в начало следующей страницы. Символ новой строки (`\n`) перемещает активную позицию в начало следующей строки. Символ возврата каретки (`\r`) переносит активную позицию в начало текущей строки. Символ горизонтальной табуляции (`\t`) перемещает активную позицию в следующую точку горизонтальной табуляции (обычно эти точки находятся в позициях 1, 9, 17, 25 и т.д.). Символ вертикальной табуляции (`\v`) переносит активную позицию в следующую точку вертикальной табуляции.

Эти управляющие последовательности не обязательно работают на всех устройствах отображения. Например, на экране ПК символы перевода страницы и вертикальной табуляции не приводят к перемещению курсора, а вызывают появление случайных символов, но они работают в соответствие с описанием, когда отправляются принтеру.

Следующие три управляющие последовательности (`\\`, `\'` и `\"`) обеспечивают возможность использования символов `\`, `'` и `"` в качестве символьных констант. (Поскольку эти символы служат для определения символьных констант как части команды `printf()`, буквальное их указание может вызвать путаницу.) Предположим, что вы хотите вывести следующую строку:

```
Джо сказал: "символ \ является символом обратной косой черты."
```

Необходимо использовать такой код:

```
printf("Джо сказал: \"символ \\ является символом обратной косой черты.\"\n");
```

Две последних формы (`\0`*oo* и `\x`*hh*) – это специальные представления ASCII-кода. Чтобы представить символ его восьмеричным ASCII-кодом, предварите код обратной косой чертой (`\`) и поместите всю конструкцию в одиночные кавычки. Например, если ваш компилятор не распознает символ предупреждения (`\a`), вы можете воспользоваться его ASCII-кодом:

```
beep = '\007';
```

Ведущие нули можно не указывать, так что запись '\07' или даже '\7' будет правильной. Эта запись вызывает интерпретацию чисел в качестве восьмеричных, даже при отсутствии начального 0.

Начиная со стандарта C90, в С доступна и третья возможность – применение шестнадцатеричной формы для символьных констант. В этом случае за символом обратной косой черты следует символ x или X и от одной до трех шестнадцатеричных цифр. Например, символу <Ctrl+P> соответствует шестнадцатеричный ASCII-код 10 (16 в десятичной форме), следовательно, его можно выразить как '\x10' или '\X010'.

На рис. 3.5 приведены примеры целочисленных констант.

Примеры целочисленных констант			
Тип	**Шестнадцатеричная**	**Восьмеричная**	**Десятичная**
char	\0x41	\0101	–
int	0x41	0101	65
unsigned int	0x41u	0101u	65u
long	0x41L	0101L	65L
unsigned long	0x41UL	0101UL	65UL
long long	0x41LL	0101LL	65LL
unsigned long long	0x41ULL	0101ULL	65ULL

Рис. 3.5. *Виды записи целочисленных констант семейства* int

При использовании кода ASCII обращайте внимание на различие между числами и символами чисел. Например, символ 4 представлен в коде ASCII значением 52. Запись '4' представляет символ 4, но не числовое значение 4.

На этом этапе у вас могут возникнуть три вопроса.

- *Почему в последнем примере управляющие последовательности не заключены в одиночные кавычки (*printf("Джо сказал: \"символ \\ является символом обратной косой черты.\"\n");*)?* Когда символ, будь он управляющей последовательностью или нет, является частью строки символов, которая заключена в двойные кавычки, не помещайте его в одиночные кавычки. Обратите внимание, что ни один из символов, использованных в этом примере (Д, ж, о и т.д.), не заключен в одиночные кавычки. Строка символов, помещенная в двойные кавычки, называется символьной строкой. (Строки рассматриваются в главе 4.) Аналогично, оператор printf("Здравствуй, мир!\007\n"); выведет строку Здравствуй, мир! и вызовет выдачу звукового сигнала, а оператор printf("Здравствуй, мир!7\n"); выведет строку Здравствуй, мир!7. Цифры, не являющиеся частью управляющей последовательности, считаются обычными символами, подлежащими выводу.

- Когда должен использоваться ASCII-код, а когда – управляющие последовательности? Если у вас есть возможность выбора между применением одной из специальных управляющих последовательностей, скажем '\f', и эквивалентного ASCII-кода, например, '\014', отдавайте предпочтение '\f'. Во-первых, при таком представлении легче понять смысл. Во-вторых, такая запись обладает лучшей переносимостью. Если вы работаете с системой, в которой не используется код ASCII, последовательность '\f' по-прежнему будет работать.

- Если нужно применять цифровой код, почему необходимо указывать, скажем, '\032', а не 032? Во-первых, использование записи '\032' вместо 032 позволит другому программисту, читающему код, понять, что вы намереваетесь представить код символа. Во-вторых, управляющая последовательность, такая как \032, может быть встроена в часть строки C тем же способом, что и \007 в первом вопросе.

Печатаемые символы

Для указания на то, что должен быть выведен символ, в функции printf() используется спецификатор %c. Вспомните, что символьная переменная хранится как однобайтовое целочисленное значение. Следовательно, при выводе значения переменной типа char с обычным спецификатором %d будет получено целое число. Спецификатор формата %c сообщает функции printf() о необходимости отобразить символ с кодовым значением, равным этому целому числу. В листинге 3.5 приведен код, в котором переменная char выводится обоими способами.

Листинг 3.5. Программа charcode.c

```
/* charcode.c -- отображает кодовое значение символа */
#include <stdio.h>
int main(void)
{
   char ch;
   printf("Введите какой-нибудь символ.\n");
   scanf("%c", &ch);   /* пользователь вводит символ */
   printf("Код символа %c равен %d.\ n", ch, ch);
   return 0;
}
```

Вот пример выполнения этой программы:

```
Введите какой-нибудь символ.
C
Код символа C равен 67.
```

При работе с программой не забывайте нажимать клавишу <Enter> или <Return> после ввода символа. Функция scanf() затем извлекает символ, введенный с клавиатуры, а амперсанд (&) означает, что этот символ присваивается переменной ch. Далее с помощью функции printf() значение переменной ch выводится два раза, сначала как символ (на что указывает спецификатор %c), а потом как десятичное целое число (на что указывает спецификатор %d). Обратите внимание, что спецификаторы функции printf() определяют способ отображения данных, но не то, как они хранятся в памяти (рис. 3.6).

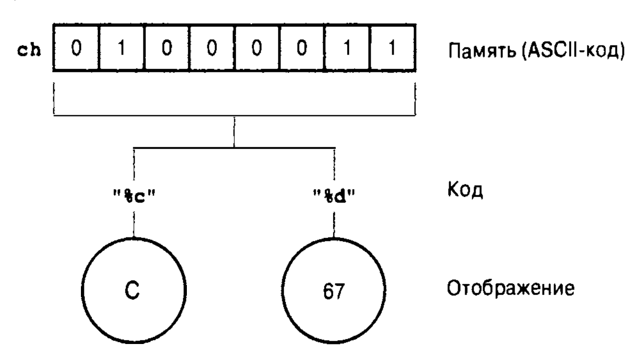


Рис. 3.6. Отображение данных на экране и их хранение в памяти

Со знаком или без знака?

В некоторых реализациях С тип char является типом со знаком. Это значит, что переменная типа char может принимать значения из диапазона от -128 до 127. В других реализациях тип char сделан беззнаковым и может иметь значения из диапазона от 0 до 255. В описании вашего компилятора должно быть явно указано, к какой разновидности принадлежит тип char, либо вы это можете узнать, заглянув в заголовочный файл limits.h, который рассматривается в следующей главе.

Согласно стандарту C90, язык С позволяет использовать ключевые слова signed и unsigned с типом char. Затем, независимо от того, какими являются данные типа char по умолчанию, тип signed char будет со знаком, а тип unsigned char — без знака. Такие версии типа char удобны, если этот тип применяется для обработки небольших целых чисел. Для собственно символов используйте стандартный тип char без модификаторов.

Тип _Bool

Тип _Bool, появившийся в стандарте C99, применяется для представления булевских значений, т.е. логических значений true (истина) и false (ложь). Поскольку в языке С для представления true используется значение 1, а для представления false — значение 0, тип _Bool по существу является целочисленным типом, но таким, который в принципе требует всего 1 бит памяти, поскольку этого достаточно, чтобы охватить весь диапазон от 0 до 1.

В программах булевские значения применяются при выборе того, какой код должен выполняться следующим. Более подробно выполнение кода описано в главах 6 и 7, поэтому дальнейшее обсуждение мы откладываем до указанных глав.

Переносимые типы: stdint.h и inttypes.h

К этому моменту вы, скорее всего, уже обратили внимание, что язык С предлагает широкое разнообразие целочисленных типов, и это можно только приветствовать. И, вероятно, вы также заметили, что одно и то же имя типа не обязательно означает одно и то же в разных системах, что не столь отрадно. Было бы замечательно, если бы язык С располагал типами, смысл которых не зависит от системы. Начиная со стандарта C99, нечто подобное было достигнуто.

Это было сделано в языке за счет создания дополнительных имен для существующих типов. Секрет в том, что эти новые имена определены в заголовочном файле stdint.h. Например, int32_t представляет тип для 32-битного целого значения со знаком. В системе, в которой используется 32-битный тип int, указанный заголовочный файл может определять int32_t в качестве псевдонима типа int. В другой системе, где применяется 16-битный тип int и 32-битный тип long, это же имя, int32_t, может быть определено как псевдоним для типа long. Тогда при создании программы с использованием int32_t в качестве типа и включении заголовочного файла stdint.h компилятор будет заменять тип int или long так, как это подходит для конкретной системы.

Рассмотренные альтернативные имена являются примерами *целочисленных типов с точной шириной*. Тип int32_t содержит в точности 32 бита — ни больше и ни меньше. Не исключено, что целевая система не поддерживает эти варианты, так что целочисленные типы с точной шириной следует считать необязательными.

А что, если система не способна поддерживать типы с точной шириной? Стандарты C99 и C11 предоставляют вторую категорию альтернативных имен, которые являются

обязательными. Этот набор имен гарантирует, что тип достаточно велик, чтобы удовлетворять спецификации, и нет других типов, которые могут выполнить нужную работу, но имеют меньший размер. Эти типы называются *типами с минимальной шириной*. Например, `int_least8_t` представляет собой псевдоним наименьшего доступного типа, который может хранить 8-битное целочисленное значение со знаком. Если бы наименьший тип в конкретной системе был 16-битным, то тип `int8_t` не определялся бы. Однако тип `int_least8_t` был бы доступным и, скорее всего, реализованным как 16-битный целочисленный тип.

Конечно, есть программисты, которых больше заботит быстродействие, чем расход памяти. Для них стандарты C99 и C11 определяют набор типов, которые обеспечат максимально быстрые вычисления. Эти типы называются *высокоскоростными типами с минимальной шириной*. Например, тип `int_fast8_t` может быть определен как альтернативное имя для целочисленного типа данных вашей системы, который обеспечивает высокоскоростные вычисления с участием 8-битных значений со знаком.

Наконец, некоторых программистов устраивает только максимально возможный в системе целочисленный тип; такому типу соответствует имя `intmax_t` и он может хранить любое допустимое целочисленное значение со знаком. Аналогично, `uintmax_t` представляет тип наибольшего допустимого целочисленного значения без знака. Кстати, указанные типы могут быть больше, чем `long long` и `unsigned long`, т.к. реализациям С разрешено определять типы, выходящие за рамки обязательных. Например, некоторые компиляторы ввели тип `long long` еще до того, как он стал частью стандарта.

Стандарты C99 и C11 не только предоставляют эти новые переносимые имена типов, но также содействуют с вводом и выводом значений таких типов. Например, функция `printf()` требует определенных спецификаторов для конкретных типов. Так что нужно сделать, чтобы отобразить значение `int32_t`, когда для одного определения может требоваться спецификатор `%d`, а для другого — `%ld`? Текущий стандарт предоставляет строковые макросы (этот механизм описан в главе 4), предназначенные для отображения переносимых типов. Например, файл заголовка `inttypes.h` определит `PRId32` в качестве строки, представляющей подходящий спецификатор (скажем, `d` или `l`) для 32-битного значения со знаком. В листинге 3.6 приведен краткий пример, иллюстрирующий применение переносимого типа и связанного с ним спецификатора. Заголовочный файл `inttypes.h` включает и файл заголовка `stdint.h`, поэтому в программе придется включать только файл `inttypes.h`.

Листинг 3.6. Программа `altnames.c`

```
/* altnames.c -- переносимые имена для целочисленных типов */
#include <stdio.h>
#include <inttypes.h> // поддерживает переносимые типы
int main(void)
{
    int32_t me32;      // me32 -- это 32-битная переменная со знаком

    me32 = 45933945;
    printf("Сначала предположим, что int32_t является int: ");
    printf("me32 = %d\n", me32);
    printf("Далее не будем делать никаких предположений.\n");
    printf("Вместо этого воспользуемся \"макросом\" из файла inttypes.h: ");
    printf("me32 = %" PRId32 "\n", me32);

    return 0;
}
```

В финальном вызове функции `printf()` аргумент `PRId32` заменяется своим определением `"d"` из файла `inttypes.h`, в результате чего строка принимает такой вид:

```
printf("me32 = %" "d" "\n", me32);
```

Однако C объединяет последовательно идущие строки в кавычках в одну строку в двойных кавычках, давая в результате следующую строку:

```
printf("me16 = %d\n", me16);
```

Ниже показан вывод программы; обратите внимание, что в рассматриваемом примере также используется управляющая последовательность \" для отображения двойных кавычек:

```
Сначала предположим, что int32_t является int: me32 = 45933945
Далее не будем делать никаких предположений.
Вместо этого воспользуемся "макросом" из файла inttypes.h: me32 = 45933945
```

В этом разделе не ставится цель изучить все расширенные целочисленные типы. Намерение скорее состоит в демонстрации наличия этого уровня управления типами на тот случай, если он потребуется. Подробное описание заголовочных файлов `inttypes.h` и `stdint.h` приведено в справочном разделе VI приложения Б.

НА ЗАМЕТКУ! Поддержка C99/C11

Несмотря на то что язык C перешел на стандарт C11, даже средства стандарта C99 разработчики компиляторов внедряли в разном темпе и с отличающимися приоритетами. На момент написания этой книги в некоторых компиляторах еще не были реализованы заголовочный файл `inttypes.h` и связанные с ним возможности.

Типы `float`, `double` и `long double`

Разнообразные целочисленные типы нормально подходят для большинства проектов по разработке программного обеспечения. Тем не менее, ориентированные на математику и финансы программы часто оперируют числами *с плавающей запятой*. В языке C такие числа имеют тип `float`, `double` или `long double`. Они соответствуют данным вещественного типа в языках программирования FORTRAN и Pascal. Как упоминалось ранее, числа с плавающей запятой позволяют представлять намного больший диапазон чисел, включая десятичные дроби. Представление чисел с плавающей запятой подобно *научной форме записи*, которая применяется для выражения очень больших и очень маленьких чисел. Давайте рассмотрим такую форму записи.

В научной форме записи числа представляются в виде десятичных чисел, умноженных на степень числа 10. Рассмотрим несколько примеров.

Число	Научная форма записи	Экспоненциальная форма записи
1 000 000 000	1.0×10^{9}	1.0e9
123 000	1.23×10^{5}	1.23e5
322,56	3.2256×10^{2}	3.2256e2
0,000056	5.6×10^{-5}	5.6e-5

В первом столбце показана обычная форма записи числа, во втором столбце — научная форма записи, а в третьем — экспоненциальная форма записи, которая представляет собой научную форму записи, обычно используемую при работе с компьютерами, при этом за обозначением *e* следует показатель степени 10. На рис. 3.7 приведено еще несколько примеров чисел с плавающей запятой.

Стандарт языка С требует, чтобы тип `float` был способен представлять минимум шесть значащих цифр и охватывал диапазон значений, по меньшей мере, от 10^{-37} до 10^{+37}. Первое требование означает, что тип `float` должен представлять, как минимум, первые шесть цифр такого числа, как `33.333333`. Второе требование по достоинству оценят те, кто оперирует такими величинами, как масса Солнца (2.0e30 килограмм), электрический заряд протона (1.6e–19 кулона) или сумма государственного долга. Часто для хранения чисел с плавающей запятой системы используют 32 бита. Восемь битов отводятся под значение экспоненты и ее знака, а остальные 24 бита служат для представления неэкспоненциальной части числа, которая называется *мантиссой* или *значащей частью числа*, и ее знака.

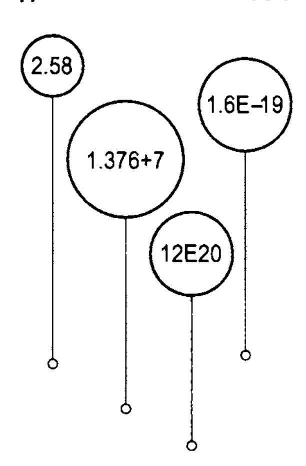


Рис. 3.7. Примеры чисел с плавающей запятой

Для представления чисел с плавающей запятой язык С предлагает также тип `double` (обеспечивающий двойную точность). Тип `double` имеет те же требования к минимальному диапазону возможных значений, что и `float`, но поддерживает более высокое минимальное количество значащих цифр – 10. В типичных представлениях типа `double` применяются 64 бита, а не 32. В некоторых системах все 32 дополнительных бита используются для неэкспоненциальной части. Это приводит к увеличению количества значащих цифр и сокращению ошибок, связанных с округлением. В других системах часть этих битов используется для размещения большей экспоненты, благодаря чему расширяется диапазон возможных значений. Любой из этих подходов обеспечивает, как минимум, 13 значащих цифр, что более чем удовлетворяет минимальному требованию стандарта.

Язык С допускает третий тип данных с плавающей запятой: `long double`. Цель этого типа – достижение большей точности, чем у типа `double`. Однако С гарантирует только то, что точность типа `long double`, по меньшей мере, не уступает точности типа `double`.

Объявление переменных с плавающей запятой

Переменные с плавающей запятой объявляются и инициализируются той же самой манере, что и переменные целочисленных типов. Ниже приведено несколько примеров:

```
float noah, jonah;
double trouble;
float planck = 6.63e-34;
long double gnp;
```

Константы с плавающей запятой (литералы)

Записывать литеральную константу с плавающей запятой можно многими способами. Основная форма записи константы с плавающей запятой выглядит как последовательность цифр со знаком, включающая десятичную точку, за которой следует буква *e* или *E* и экспонента со знаком, представляющая степень числа 10. Вот два примера допустимых констант с плавающей запятой:

```
-1.56E+12
2.87e-3
```

Знак "плюс" можно не указывать. Можно также опустить десятичную точку (2E5) или экспоненциальную часть (19.28), но не то и другое одновременно. Можно обой-

тись без дробной части (3.E16) или целой части (.45E-6), но не без обоих компонентов сразу. Ниже показано еще несколько допустимых констант с плавающей запятой:

```
3.14159
.2
4e16
.8E-5
100.
```

Не применяйте пробелы в константах с плавающей запятой. Например, эта константа является недопустимой:

```
1.56 E+12
```

По умолчанию компилятор предполагает, что константы с плавающей запятой имеют тип `double`. Предположим, например, что `some` является переменной типа `float`, и есть следующий оператор:

```
some = 4.0 * 2.0;
```

Тогда значения `4.0` и `2.0` сохраняются как данные типа `double` с использованием для каждого (обычно) 64 бита. Произведение вычисляется с применением арифметики с двойной точностью, и только после этого результат усекается к обычному типу `float`. Это гарантирует более высокую точность вычислений, но замедляет выполнение программы.

Язык С позволяет переопределить такое стандартное поведение компилятора за счет использования суффикса `f` или `F`, который заставляет компилятор трактовать константу с плавающей запятой как тип `float`, например, `2.3f` и `9.11E9F`. Суффикс `l` или `L` определяет число типа `long double`, например, `54.3l` и `4.32e4L`. Отметим, что букву `L` труднее перепутать с `1` (единица), чем букву `l`. Если число с плавающей запятой не содержит суффикса, оно относится к типу `double`.

Начиная со стандарта C99, в языке С имеется новый формат для выражения констант с плавающей запятой. В нем применяется шестнадцатеричный префикс (`0x` или `0X`) с шестнадцатеричными цифрами, `p` или `P` вместо `e` или `E` и экспонента, которая является степенью 2, а не 10. Такое число может выглядеть следующим образом:

```
0xa.1fp10
```

`a` — это 10 в шестнадцатеричной системе, `.1f` — $^{1}/_{16}$ плюс $^{15}/_{256}$) (`f` — это 15 в шестнадцатеричной системе), а `p10` — это 2^{10}, или 1024. В результате полное значение равно $(10 + ^{1}/_{16} + ^{15}/_{256}) \times 1024$, или 10364,0 в десятичной форме записи.

Не все компиляторы С поддерживают эту функциональную возможность.

Вывод значений с плавающей запятой

Функция `printf()` использует спецификатор формата `%f` для вывода чисел типа `float` и `double` в десятичном представлении и спецификатор `%e` для вывода в экспоненциальном представлении. Если ваша система поддерживает шестнадцатеричный формат чисел с плавающей запятой, то вместо `e` или `E` можно применять `a` или `A`. Для вывода данных типа `long double` требуются спецификаторы `%Lf`, `%Le` и `%La`. Обратите внимание, что для вывода как `float`, так и `double` используется спецификатор `%f`, `%e` или `%a`. Причина в том, что язык С автоматически расширяет значения `float` до типа `double`, когда они передаются в качестве аргументов любой функции, такой как `printf()`, в прототипе которой тип аргумента не определен явным образом. Это поведение демонстрируется в листинге 3.7.

Листинг 3.7. Программа showf_pt.c

```
/* showf_pt.c -- отображает значение типа float двумя способами */
#include <stdio.h>
int main(void)
{
    float aboat = 32000.0;
    double abet = 2.14e9;
    long double dip = 5.32e-5;

    printf("%f может быть записано как %e\n", aboat, aboat);
    // для вывода следующей строки требуется компилятор,
    // поддерживающий C99 или более поздний стандарт
    printf("И его %a в шестнадцатеричной, представляющей степени 2, форме записи\n", aboat);
    printf("%f может быть записано как %e\n", abet, abet);
    printf("%Lf может быть записано как %Le\n", dip, dip);

    return 0;
}
```

Ниже приведен вывод, при условии, что компилятор совместим со стандартом C99/C11:

```
32000.000000 может быть записано как 3.200000e+04
И его 0x1.f4p+14 в шестнадцатеричной, представляющей степени 2, форме записи
2140000000.000000 может быть записано как 2.140000e+09
0.000053 может быть записано как 5.320000e-05
```

Этот пример иллюстрирует стандартный вывод. В следующей главе мы обсудим, как управлять внешним видом вывода путем установки ширины поля и количества позиций справа от десятичной точки.

Переполнение и потеря значимости в операциях с плавающей запятой

Предположим, что наибольшее возможное значение типа `float` равно примерно 3.4E38, и нужно выполнить следующие операции:

```
float toobig = 3.4E38 * 100.0f;
printf("%e\n", toobig);
```

Что произойдет? Это пример *переполнения*, когда в результате вычислений получается слишком большое число, чтобы его можно было правильно представить. Поведение системы в таких случаях обычно не определено, но в рассматриваемой ситуации переменной `toobig` присваивается специальное значение, которое обозначает *бесконечность*, и функция `printf()` отображает либо `inf`, либо `infinity` (или какую-то другую вариацию на эту тему).

А что можно сказать о делении очень малых чисел? Здесь ситуация более сложная. Вспомните, что число типа `float` хранится в виде сочетания показателя степени и значащей части, или *мантиссы*. В рассматриваемом случае это число, имеющее минимально возможный показатель степени, а также наименьшее значение, которое использует все доступные биты, отведенные для представления мантиссы. Это будет наименьшее число, представленное с наибольшей точностью, доступной для типа `float`. Теперь разделим его на 2. Обычно это приводит к уменьшению показателя степени, но в данном случае показатель уже достиг своего нижнего предела. Происходит сдвиг битов мантиссы вправо, с освобождением первой позиции и потерей последней двоичной цифры. Аналогичная картина возникает, если выбрать 10 в качестве основания системы счисления, взять число с четырьмя значащими цифрами, например, 0.1234E-10, и разделить его на 10, получив в итоге 0.0123E-10. Вы получите результат,

но в процессе деления потеряете цифру. Эта ситуация называется потерей *значимости*, а значения с плавающей запятой, которые утратили полную точность типа, в языке С называются *субнормальными*. Таким образом, деление наименьшего положительного значения с плавающей запятой на 2 дает субнормальное значение. Деление на достаточно большое значение приведет к потере всех цифр, и вы получите в результате 0. В настоящее время библиотека С предоставляет функции, которые позволяют проверить, не приведут ли вычисления к субнормальным значениям.

Существует еще одно специальное значение с плавающей запятой: NaN (not-a-number – не число). Например, вы передаете функции asin() некоторое значение, а она возвращает угол, для которого переданное значение является синусом. Однако значение синуса не может быть больше 1, поэтому функция не определена для значений, превышающих 1. В таких случаях функция возвращает значение NaN, которое функция printf() отображает в виде nan, NaN или каким-то похожим образом.

Ошибки округления данных с плавающей запятой

Возьмите некоторое число, прибавьте к нему 1 и затем отнимите от суммы исходное число. Что вы получите в результате? Вы получите 1. Тем не менее, вычисления с плавающей запятой вроде показанного ниже могут дать другой результат:

```
/* floaterr.c -- демонстрирует ошибку округления */
#include <stdio.h>
int main(void)
{
   float a,b;
   b = 2.0e20 + 1.0;
   a = b - 2.0e20;
   printf("%f \n", a);
   return 0;
}
```

Вывод выглядит следующим образом:

```
0.000000          ← старая версия компилятора gcc в операционной системе Linux
-13584010575872.000000 ←Turbo C 1.5
4008175468544.000000   ←XCode 4.5, Visual Studio 2012,
                        текущая версия компилятора gcc
```

Причина получения таких странных результатов состоит в том, что компьютер не следит за тем, чтобы под числа с плавающей запятой было отведено столько десятичных позиций, сколько нужно для правильного выполнения операции. Число 2.0e20 представлено цифрой 2, за которой следует 20 нулей, и за счет прибавления 1 вы пытаетесь изменить 21-ю цифру. Чтобы эта операция выполнилась корректно, программа должна иметь возможность хранить число, состоящее из 21 цифры. Число типа float — это обычно шесть или семь цифр, масштабированных при помощи показателя степени до большего или меньшего числа, так что такая попытка сложения обречена на неудачу. С другой стороны, если вместо 2.0e20 вы укажете 2.0e4, то получите правильный ответ, поскольку вы пытаетесь изменить пятую цифру, а числа типа float обладают достаточной для этой операции точностью.

Представление значений с плавающей запятой

В предыдущей врезке было видно, что вывод одной и той же программы отличался в зависимости от используемой компьютерной системы. Причина этого отличия в том, что существует много разных способов реализации представления чисел с плавающей запятой в рамках описанных общих подходов. Для обеспечения большего единообразия в Институте инженеров по электротехнике и радиоэлектронике (IEEE) разработан стандарт для представления чисел с плавающей запятой и вычислений с плавающей запятой, который теперь применяется во многих аппаратных блоках обработки чисел с плавающей запятой.

В 2011 году этот стандарт был принят в качестве международного стандарта ISO/IEC/IEEE 60559:2011. Он вошел в качестве необязательной части в стандарты C99 и C11, исходя из предположения, что его будут поддерживать платформы с соответствующим оборудованием. Последний пример вывода программы floaterr.c получен в системе, которая поддерживает этот стандарт для представления чисел с плавающей запятой. Поддержка со стороны языка С включает инструменты для выявления описанной проблемы. Более подробные сведения приведены в разделе V приложения Б.

Комплексные и мнимые типы

Во многих научных и технических расчетах используются комплексные и мнимые числа. Стандарт C99 поддерживает эти числа, но с некоторыми оговорками. В автономной реализации, такой как применяемая для встроенных процессоров, необходимость в наличии этих типов отсутствует. (Заметим, что микропроцессор видеомагнитофона при выполнении своей задачи вряд ли нуждается в комплексных числах.) Также в общем случае мнимые типы являются необязательными. В стандарте C11 весь пакет поддержки комплексных чисел сделан необязательным.

Существуют три комплексных типа, названные float _Complex, double _Complex и long double _Complex. Переменная типа float _Complex, к примеру, будет содержать два значения float, одно из которых представляет действительную часть комплексного числа, а другое — его мнимую часть. Аналогично, существуют три мнимых типа: float _Imaginary, double _Imaginary и long double _Imaginary.

Включение заголовочного файла complex.h делает возможной подстановку слова complex взамен _Complex и слова imaginary взамен _Imaginary, а также применение символа I для представления квадратного корня из -1.

Вас может интересовать, а почему в стандарте языка С просто не ввели complex в качестве ключевого слова вместо того, чтобы использовать _Complex и затем включать заголовочный файл для определения complex как подстановки взамен _Complex. Комитет по стандартизации обычно не спешит вводить новое ключевое слово, поскольку это может привести к утере допустимости существующего кода, в котором такое слово применялось в качестве идентификатора.

Например, до появления стандарта C99 многие программисты уже использовали struct complex для представления комплексных чисел или, возможно, психологических состояний. (Ключевое слово struct, как объясняется в главе 14, служит для определения структур данных, способных хранить более одного значения.) Превращение слова "complex" в ключевое слово превратило бы предшествующие случаи его применения в синтаксические ошибки. С другой стороны, использование сочетания struct _Complex значительно менее вероятно, особенно с учетом того, что идентификаторы с начальным символом подчеркивания считаются зарезервированными. Таким образом, комитет остановился на _Complex в качестве ключевого слова и сделал вариант complex доступным для тех, кому не нужно беспокоиться по поводу конфликтов с предшествующими применениями.

За пределами базовых типов

На этом список фундаментальных типов данных завершен. Одним он может показаться слишком длинным. Другие могут посчитать, что необходимы дополнительные типы. Как насчет типа символьной строки? В языке С нет такого типа, но он мог бы обеспечить удобную работу со строками. Первое представление о строках вы получите в главе 4.

В С имеются другие типы, производные от базовых типов. Они включают массивы, указатели, структуры и объединения. Хотя эти типы являются предметом обсуждения последующих глав, кое-что уже было вскользь затронуто в примерах настоящей главы. Например, *указатель* указывает на место в памяти, в котором хранится переменная или другой объект данных. Префикс &, используемый в функции scanf(), создает указатель, который сообщает функции, куда помещать информацию.

Сводка: базовые типы данных

Ключевые слова

Базовые типы данных устанавливаются с применением 11 ключевых слов: int, long, short, unsigned, char, float, double, signed, _Bool, _Complex и _Imaginary.

Целые числа со знаком

Они могут иметь как положительные, так и отрицательные значения.

- int — базовый целочисленный тип в заданной системе. Язык С гарантирует для int не менее 16 битов.
- short или short int — максимальное целое число типа short не превосходит наибольшего целочисленного значения типа int. Язык С гарантирует для short не менее 16 битов.
- long или long int — может хранить целое число, которое, как минимум, не меньше наибольшего числа типа int или больше его. Язык С гарантирует для long не менее 32 битов.
- long long или long long int — этот тип может быть целым числом, которое, как минимум, не меньше наибольшего числа типа long, а, возможно, и больше его. Для long long гарантируются не менее 64 битов.

Обычно тип long имеет большую длину, чем short, а длина типа int совпадает с длиной одного из этих типов. Например, старые основанные на DOS системы для IBM PC предоставляли 16-битные типы short и int и 32-битный тип long, а позже системы, основанные на Windows 95, предлагали 16-битный тип short и 32-битные типы int и long.

При желании для любого типа со знаком можно указывать ключевое слово signed, делая явным тот факт, что этот тип поддерживает знак.

Целые без знака

Такие типы хранят только нулевое или положительные значения. Это расширяет диапазон до большего возможного положительного числа. Указывайте ключевое слово unsigned перед желаемым типом: unsigned int, unsigned long, unsigned short. Одиночное ключевое unsigned означает то же, что и unsigned int.

Символы

Ими являются типографские символы, такие как A, & и +. По определению тип char для представления символа использует 1 байт памяти. Исторически сложилось так, что байт символа чаще всего имеет длину 8 битов, но он может быть длиной 16 битов или больше, если это необходимо для представления базового набора символов.

- char — ключевое слово для этого типа. В одних реализациях применяется тип char со знаком, в других он без знака. Язык С позволяет использовать ключевые слова signed и unsigned для указания нужной формы.

Булевские значения

Булевский тип представляет значения `true` (истина) и `false` (ложь); в языке C для представления `true` применяется `1`, а для представления `false` — `0`.

- `_Bool` — ключевое слово для этого типа. Он является типом `int` без знака и должен быть настолько большим, чтобы обеспечить хранение значений из диапазона от 0 до 1.

Вещественные числа с плавающей запятой

Эти типы могут иметь как положительные, так и отрицательные значения.

- `float` — базовый тип данных с плавающей запятой в системе; он может представлять, по меньшей мере, шесть значащих цифр с заданной точностью.
- `double` — (возможно) большая единица для хранения чисел с плавающей запятой. Этот тип может разрешать большее количество значащих цифр (минимум 10, но обычно больше) и, возможно, большие значения показателя степени, чем тип `float`.
- `long double` — (возможно) еще большая единица для хранения чисел с плавающей запятой. Этот тип может разрешать большее количество значащих цифр и, возможно, большие значения показателя степени, чем тип `double`.

Комплексные и мнимые числа с плавающей запятой

Мнимые типы являются необязательными. Вещественные и мнимые компоненты основаны на соответствующих вещественных типах:

- `float _Complex`
- `double _Complex`
- `long double _Complex`
- `float _Imaginary`
- `double _Imaginary`
- `long double _Imaginary`

Сводка: объявление простой переменной

1. Выберите необходимый тип данных.
2. Выберите имя для переменной, включающее разрешенные символы.
3. Применяйте следующий формат для оператора объявления:

   спецификатор-типа имя-переменной;

   Компонент *спецификатор-типа* образуется из одного или большего количества ключевых слов для типов; вот примеры объявлений:

   ```
   int erest;
   unsigned short cash;
   ```

4. Вы можете объявлять сразу несколько переменных одного и того же типа, разделяя имена переменных запятыми. Например:

   ```
   char ch, init, ans;
   ```

5. Вы можете инициализировать переменную в операторе объявления:

   ```
   float mass = 6.0E24;
   ```

Размеры типов

Какие размеры типов используются в вашей системе? Чтобы выяснить это, попробуйте выполнить программу, показанную в листинге 3.8.

Листинг 3.8. Программа typesize.c

```
/* typesize.c -- выводит размеры типов */
#include <stdio.h>
int main(void)
{
    /* В стандарте c99 для размеров предусмотрен спецификатор %zd */
    printf("Тип int имеет размер %zd байт(ов).\n", sizeof(int));
    printf("Тип char имеет размер %zd байт(ов).\n", sizeof(char));
    printf("Тип long имеет размер %zd байт(ов).\n", sizeof(long));
    printf("Тип long long имеет размер %zd байт(ов).\n",
           sizeof(long long));
    printf("Тип double имеет размер %zd байт(ов).\n",
           sizeof(double));
    printf("Тип long double имеет размер %zd байт(ов).\n",
           sizeof(long double));
    return 0;
}
```

В языке C имеется встроенная операция `sizeof`, которая возвращает размер типа в байтах. Для такого применения `sizeof` в стандартах C99 и C11 предоставляется спецификатор `%zd`. Компиляторы, несовместимые с этими стандартами, могут потребовать вместо него спецификатор `%u` или `%lu`. Ниже показан пример вывода программы `typesize.c`:

```
Тип int имеет размер 4 байт(ов).
Тип char имеет размер 1 байт(ов).
Тип long имеет размер 8 байт(ов).
Тип long long имеет размер 8 байт(ов).
Тип double имеет размер 8 байт(ов).
Тип long double имеет размер 16 байт(ов).
```

Эта программа находит размеры только шести типов, но вы легко можете ее модифицировать, чтобы она определяла размер любого другого интересующего типа. Обратите внимание, что размером типа `char` обязательно будет 1 байт, потому что в языке C размер одного байта определяется в терминах `char`. Таким образом, в системе с 16-битным типом `char` и 64-битным `double` операция `sizeof` сообщит, что тип `double` имеет размер 4 байта. Для получения более подробной информации о предельных размерах типов можете просмотреть заголовочные файлы `limits.h` и `float.h`. (Эти два файла обсуждаются в следующей главе.)

Кстати, в нескольких последних строках обратите внимание на то, что оператор `printf()` можно разнести на две строки. Его можно разделять на большее количество частей при условии, что разрыв не происходит внутри раздела, заключенного в кавычки, или в середине слова.

Использование типов данных

При разработке программы обращайте внимание на то, какие переменные необходимы, и какие типы они должны иметь. Скорее всего, для чисел вы выберете `int` или, возможно, `float`, а для символов – тип `char`. Объявляйте переменные в начале фун-

кции, в которой они используются. Выбирайте для переменных имена, отражающие их предназначение. При инициализации обеспечьте соответствие типов констант и типов переменных. Например:

```
int apples = 3;       /* правильно */
int oranges = 3.0;    /* плохая форма */
```

В отношении несовпадения типов язык С более либерален, чем, скажем, Pascal. Компиляторы С разрешают инициализацию, сделанную во втором операторе, но могут выдать сообщение, особенно если установлен высокий уровень предупреждений. Поэтому лучше не вырабатывать в себе плохие привычки.

Когда вы инициализируете переменную одного числового типа значением другого числового типа, компилятор С преобразует такое значение в тип переменной. Это означает возможность потери данных. Например, взгляните на следующие примеры инициализации:

```
int cost = 12.99;      /* инициализация переменной типа int значением double */
float pi = 3.1415926536; /* инициализация переменной типа float значением double */
```

В первом объявлении переменной cost присваивается значение 12; при преобразовании значений с плавающей запятой в целочисленные компилятор С вместо округления просто отбрасывает дробную часть числа (выполняет *усечение*). Во втором объявлении происходит некоторая потеря точности, поскольку для типа float точность гарантируется только в пределах шести цифр. Когда вы делаете такую инициализацию, компиляторы могут (но не обязаны) выдавать предупреждающее сообщение. Подобная проблема могла возникнуть при компиляции программы, представленной в листинге 3.1.

Многие программисты и организации придерживаются систематических соглашений по назначению имен переменным, согласно которым имя отражает тип переменной. Например, можно было бы воспользоваться префиксом i_ для указания типа int и префиксом us_ для отражения типа unsigned short, так что i_smart немедленно опознается как переменная типа int, а us_verysmart — как переменная типа unsigned short.

Аргументы и связанные с ними ловушки

Полезно еще раз повторить и акцентировать внимание на сделанном ранее в этой главе предупреждении, касающемся использования функции printf(). Как вы можете помнить, элементы информации, передаваемой функции, называются *аргументами*. К примеру, вызов функции printf("Здравствуй, мир.") содержит один аргумент: "Здравствуй,мир.". Последовательность символов в кавычках вроде "Здравствуй, мир." называется *строкой*. Строки будут обсуждаться в главе 4. Пока что важным моментом является то, что строка, даже если она содержит несколько слов и знаков препинания, считается одним аргументом.

Аналогично, вызов функции scanf("%d", &weight) содержит два аргумента: "%d" и &weight. Для отделения аргументов друг от друга в языке С применяются запятые. Функции printf() и scanf() необычны в том, что они не ограничены конкретным количеством аргументов. Например, мы вызывали printf() с одним, двумя и тремя аргументами. Чтобы программа работала должным образом, она должна знать, сколько аргументов получает функция. Функции printf() и scanf() используют первый аргумент для указания количества дополнительных аргументов, который будут переданы. Дело в том, что каждая спецификация формата в первой строке говорит о наличии дополнительного аргумента.

Например, приведенный ниже оператор содержит два спецификатора формата, %d и %d:

```
printf("%d котов съедают %d банок тунца\n", cats, cans);
```

Это сообщает о том, что функция должна ожидать еще два аргумента, и действительно, дальше следуют два аргумента — cats и cans.

Как программист, вы отвечаете за гарантию того, что количество спецификаций формата соответствует числу дополнительных аргументов, а типы спецификаторов соответствуют типам значений. В настоящее время язык С располагает механизмом прототипирования функций, который проверяет правильность количества и типов аргументов в вызове функции, однако он не работает в случае функций printf() и scanf(), т.к. они принимают переменное число аргументов. Что случится, если программист не справится со своей обязанностью по отношению к аргументам? Предположим, вы написали программу, показанную в листинге 3.9.

Листинг 3.9. Программа badcount.c

```
/* badcount.c -- некорректное количество аргументов */
#include <stdio.h>
int main(void)
{
    int n = 4;
    int m = 5;
    float f = 7.0f;
    float g = 8.0f;

    printf("%d\n", n, m);     /* слишком много аргументов   */
    printf("%d %d %d\n", n);  /* аргументов недостаточно    */
    printf("%d %d\n", f, g);  /* неправильные типы значений */

    return 0;
}
```

Ниже приведен пример вывода, полученный в случае применения компилятора XCode 4.6 (OS X 10.8):

```
4
4 1 -706337836
1606414344 1
```

А так выглядит вывод этой же программы при использовании Microsoft Visual Studio Express 2012 (Windows 7):

```
4
4 0 0
0 1075576832
```

Обратите внимание, что применение спецификатора %d для отображения значения float не приводит к преобразованию значения float в ближайшее значение int. Кроме того, результаты, которые вы получаете при недостаточном количестве аргументов или в случае указания некорректных их типов, отличаются от платформы к платформе и от запуска к запуску.

Ни один из опробованных нами компиляторов не отказался компилировать этот код, хотя большинство из них выдавало предупреждения о том, что впоследствии могут возникнуть проблемы. Не было также никаких жалоб во время выполнения программы. Действительно, некоторые компиляторы могут обнаруживать ошибки подобного рода, но стандарт языка С вовсе не требует этого. Следовательно, компилятор

может не выявить таких ошибок, а поскольку во всех других отношениях программа, возможно, ведет себя корректно, то есть шанс вообще не заметить ошибки. Если программа не выводит ожидаемое количество значений или выводит неожиданные значения, проверьте, указано ли правильное число аргументов в вызове функции printf().

Еще один пример: управляющие последовательности

Давайте рассмотрим еще один пример, связанный с выводом, в котором используются специальные управляющие последовательности для символов языка С. В частности, программа, представленная в листинге 3.10, демонстрирует работу символов возврата на одну позицию влево (\b), табуляции (\t) и возврата каретки (\r). Их концепции существуют со времен, когда компьютеры применяли для вывода телетайпы, и они не всегда успешно транслируются в современных графических интерфейсах. Например, код в листинге 3.10 не работает описанным здесь образом в некоторых реализациях для компьютеров Macintosh.

Листинг 3.10. Программа escape.c

```
/* escape.c -- использование управляющих последовательностей */
#include <stdio.h>
int main(void)
{
   float salary;

   printf("\aВведите желаемую сумму месячной зарплаты:");    /* 1 */
   printf(" $_______\b\b\b\b\b\b\b");                         /* 2 */
   scanf("%f", &salary);
   printf("\n\t$%.2f в месяц соответствует $%.2f в год.", salary,
          salary * 12.0);                                     /* 3 */
   printf("\rOro!\n");                                        /* 4 */

   return 0;
}
```

Результаты выполнения программы

Давайте пошагово пройдемся по этой программе и посмотрим, как она будет работать в системе, где управляющие последовательности ведут себя описанным образом. (Фактическое поведение может отличаться. Например, XCode 4.6 отображает символы \a, \b и \r в виде перевернутых вопросительных знаков!)

Первый оператор printf() (помечен номером 1) воспроизводит звуковой сигнал (вызванный последовательностью \a), а затем выводит следующую фразу:

```
Введите желаемую сумму месячной зарплаты:
```

Поскольку в конце строки отсутствует последовательность \n, курсор устанавливается в позицию, следующую за двоеточием.

Второй оператор printf() начинает вывод с позиции, где остановился первый оператор, поэтому после его выполнения вывод на экране выглядит так:

```
Введите желаемую сумму месячной зарплаты: $_______
```

Пробел между двоеточием и знаком доллара появился в связи с тем, что строка во втором операторе начинается с пробела. Результатом семи символов возврата на

одну позицию влево будет перемещение курсора на семь позиций влево. Курсор проходит через семь символов подчеркивания и располагается непосредственно после знака доллара. Обычно при возврате на одну позицию влево символы, через которые проходит курсор, не очищаются, но в некоторых реализациях может применяться деструктивный возврат на одну позицию (т.е. забой), поэтому результаты выполнения данной простой программы изменятся.

В этом месте вы вводите с клавиатуры свой ответ, скажем, `4000.00`. Теперь строка принимает следующий вид:

```
Введите желаемую сумму месячной зарплаты: $4000.00
```

Символы, которые вы набираете на клавиатуре, заменяют символы подчеркивания, и после нажатия клавиши <Enter> (или <Return>), чтобы ввести ответ, курсор переместится в начало следующей строки.

Вывод третьего оператора `printf()` начинается с `\n\t`. Символ новой строки перемещает курсор в начало следующей строки. Символ табуляции перемещает курсор в следующую позицию табуляции в этой строке — обычно, но не обязательно, в позицию 9. Затем выводится оставшаяся часть строки. После выполнения этого оператора экран выглядит так:

```
Введите желаемую сумму месячной зарплаты: $4000.00
        $4000.00 в месяц соответствует $48000.00 в год.
```

Поскольку в этом операторе `printf()` символ новой строки не используется, курсор остается непосредственно после завершающей точки.

Четвертый оператор `printf()` начинается с последовательности `\r`. Она помещает курсор в начало текущей строки. Затем отображается строка "Ого!" и последовательность `\n` переводит курсор на следующую строку.

Окончательный вывод на экране имеет следующий вид:

```
Введите желаемую сумму месячной зарплаты: $4000.00
Ого!    $4000.00 в месяц соответствует $48000.00 в год.
```

Сброс буфера вывода

Когда функция `printf()` действительно отправляет вывод на экран? Первоначально операторы `printf()` пересылают выходные данные в промежуточную область хранения, называемую *буфером*. Время от времени данные, находящиеся в буфере, отправляются на экран. Стандартные правила С относительно того, когда пересылать вывод из буфера на экран, довольно очевидны: он передается на экран, когда буфер заполнен, когда встречается символ новой строки или когда наступает время ввода данных. (Отправка вывода из буфера на экран или в файл называется *сбросом буфера*.) Например, первые два оператора `printf()` не заполняют буфер и не содержат символа новой строки, но непосредственно за ними следует оператор `scanf()`, который запрашивает ввод. Это инициирует отправку вывода `printf()` на экран.

Вы можете столкнуться со старой реализацией, в которой оператор `scanf()` не обеспечивает принудительную очистку буфера. Это в результате приводит к тому, что программа начинает искать введенные данные, даже предварительно не выводя на экран приглашение на ввод. В таком случае для сброса буфера можно воспользоваться символом новой строки. Код изменяется следующим образом:

```
printf("Введите желаемую сумму месячной зарплаты:\n");
scanf("%f", &salary);
```

Этот код работает независимо от того, приводит предстоящий ввод данных к сбросу буфера или нет. Однако он устанавливает курсор на следующей строке, не позволяя вводить данные в той же строке, где находится приглашение. Другое решение предусматривает применение функции `fflush()`, которая описана в главе 13.

Ключевые понятия

В языке С имеется поразительное количество числовых типов. Это отражает намерение разработчиков языка С избегать препятствий на пути программиста. Вместо заявления о том, что одного вида целочисленных значений вполне достаточно, в языке С попытались предоставить программистам возможность выбора конкретной вариации (со знаком или без) и размера типа, которые лучше всего удовлетворяют нуждам разрабатываемой программы.

В компьютере числа с плавающей запятой фундаментально отличаются от целых чисел. Они хранятся и обрабатываются по-разному. Две 32-битных единицы памяти могут содержать идентичные наборы битов, но если одна из них интерпретируется как `float`, а другая как `long`, то они будут представлять совершенно разные и не связанные между собой значения. Например, если взять в IBM PC последовательность битов, представляющую число 256.0 типа `float`, и интерпретировать его как значение типа `long`, вы получите 113246208. Язык С позволяет записывать выражения со смешанными типами данных, но будет выполнять автоматические преобразования, чтобы в действительном вычислении участвовал только один тип данных.

В памяти компьютера символы представлены числовым кодом. В США наибольшее распространение получил код ASCII, но язык С поддерживает использование и других кодов. Символьная константа – это символьное представление числового кода, применяемого в компьютерной системе; она состоит из символов, заключенных в одиночные кавычки, например, `'A'`.

Резюме

В языке С имеется большое разнообразие типов данных. Базовые типы данных разделены на две категории: целочисленные типы данных и данные с плавающей запятой. Двумя отличительными особенностями целочисленных типов являются объем памяти, выделяемой для типа, и наличие или отсутствие знака. Наименьший тип целочисленных данных – `char`, который в зависимости от реализации может быть со знаком или без знака. Можно использовать `signed char` и `unsigned char`, чтобы явно указать, какой вариант нужен, однако обычно это делается в случае использования типа `char` для хранения небольших целых чисел, а не символьных кодов. К другим целочисленным типам относятся `short`, `int`, `long` и `long long`. В языке С гарантируется, что каждый из этих типов имеет, по крайней мере, такой же размер, как предшествующий тип. Все они являются типами со знаком, но можно применять ключевое слово `unsigned` для создания соответствующих типов без знака: `unsigned short`, `unsigned int`, `unsigned long` и `unsigned long long`. Или же можно добавить модификатор `signed`, чтобы явно указать, что тип имеет знак. Наконец, существует еще тип `_Bool` – тип без знака, который способен принимать значения `0` и `1`, представляющие `false` и `true`.

Существуют три типа с плавающей запятой: `float`, `double` и, начиная со стандарта C90, `long double`. Каждый из них минимум не меньше предыдущего типа. Дополнительно реализация может поддерживать комплексные и мнимые типы за счет использования ключевых слов `_Complex` и `_Imaginary` в сочетании с ключевыми сло-

вами для типов с плавающей запятой. Например, можно работать с типами double _Complex и float _Imaginary.

Целые числа могут быть выражены в десятичной, восьмеричной и шестнадцатеричной форме. Префикс 0 указывает на восьмеричное число, а префикс 0x или 0X — на шестнадцатеричное. Например, 32, 040 и 0x20 — это десятичное, восьмеричное и шестнадцатеричное представление одного и того же значения. Суффикс l или L указывает, что значение имеет тип long, а ll или LL — что оно относится к типу long long.

Символьные константы представляются путем помещения символа в одиночные кавычки, например, 'Q', '8' и '$'. С помощью управляющих последовательностей, таких как '\n', задаются определенные непечатаемые символы. Вы можете применять форму '\007' для представления символа в коде ASCII.

Числа с плавающей запятой могут быть записаны в форме с фиксированной десятичной точкой, например, 9393.912, или в экспоненциальном представлении, например, 7.38E10. Стандарты C99 и C11 предоставляют третью форму экспоненциальной записи с использованием шестнадцатеричных цифр и степеней 2, подобно 0xa.1fp10.

Функция printf() позволяет выводить значения различных типов с применением спецификаторов, которые в своей простейшей форме состоят из знака процента и буквы, указывающей тип, например, %d или %f.

Вопросы для самоконтроля

Ответы на эти вопросы находятся в приложении А.

1. Какие типы вы будете использовать для каждого из следующих типов данных (в некоторых случаях могут подойти несколько типов данных)?

   а. Население Москвы.

   б. Стоимость копии фильма на DVD-диске.

   в. Буква, которая чаще других встречается в данной главе.

   г. Количество раз, сколько эта буква встречается в данной главе.

2. В каких случаях следует использовать переменную типа long вместо int?

3. Какие переносимые типы можно использовать, чтобы получить 32-битное целое число со знаком? Приведите аргументы в пользу своего выбора.

4. Идентифицируйте тип и назначение каждой из следующих констант:

   а. '\b'

   б. 1066

   в. 99.44

   г. 0XAA

   д. 2.0e30

5. Кое-кто написал программу с ошибками. Найдите эти ошибки.

```
include <stdio.h>
main
(
  float g; h;
  float tax, rate;
  g = e21;
  tax = rate*g;
)
```

6. Идентифицируйте тип данных (по тому, как он используется в операторах объявления) и спецификатор формата `printf()` для каждой из следующих констант.

Константа	Тип	Спецификатор
а. `12`		
б. `0X3`		
в. `'C'`		
г. `2.34E07`		
д. `'\040'`		
е. `7.0`		
ж. `6L`		
з. `6.0f`		
и. `0x5.b6p12`		

7. Определите тип данных (по тому, как он используется в операторах объявления) и спецификатор формата `printf()` для каждой из следующих констант (предполагая, что тип `int` является 16-битным).

Константа	Тип	Спецификатор
а. `012`		
б. `2.9e05L`		
в. `'s'`		
г. `100000`		
д. `'\n'`		
е. `20.0`		
ж. `0x44`		
з. `-40`		

8. Предположим, что программа начинается со следующих объявлений:

```
int imate = 2;
long shot = 53456;
char grade = 'A';
float log = 2.71828;
```

Вставьте нужные спецификаторы типа в следующие операторы `printf()`:

```
printf("Шансы попасть в %__ были %__ к 1.\n", imate, shot);
printf("Счет %__ не соответствует уровню %__.\n", log, grade);
```

9. Предположим, что `ch` является переменной типа `char`. Покажите, как присвоить ей символ возврата каретки, используя управляющую последовательность, десятичное значение, восьмеричную символьную константу и шестнадцатеричную символьную константу. (Предположите, что применяются значения кода ASCII.)

10. Исправьте следующую нелепую программу. (В языке C символом / обозначается операция деления.)

```
void main(int) / эта программа безупречна /
{
  cows, legs integer;
  printf("Сколько коровьих ног вы насчитали?\n);
  scanf("%c", legs);
  cows = legs / 4;
  printf("Отсюда следует, что есть %f коров(а).\n", cows)
}
```

11. Определите, что представляет каждая из следующих управляющих последовательностей:

    а. `\n`

    б. `\\`

    в. `\"`

    г. `\t`

Упражнения по программированию

1. Экспериментальным путем выясните, как ваша система обрабатывает переполнение при выполнении операций над целыми числами и над числами с плавающей запятой, а также потерю значимости при выполнении операций над числами с плавающей запятой; т.е. напишите программу, в которой присутствуют такие проблемы. (Для получения сведений о наибольших и наименьших значениях просмотрите обсуждение `limits.h` и `float.h` в главе 4.)
2. Напишите программу, которая приглашает ввести некоторое значение в коде ASCII, например, 66, а затем выводит символ, которому соответствует введенный код.
3. Напишите программу, которая выдает предупредительный звуковой сигнал, а затем выводит следующий текст:

```
Напуганная внезапным звуком, Вика вскрикнула:
"Во имя всех звезд, что это было!"
```

4. Напишите программу, которая считывает число с плавающей запятой и выводит его сначала в десятичном представлении, потом в экспоненциальном представлении и затем в двоично-экспоненциальном представлении, если система его поддерживает. Вывод должен быть представлен в следующем формате (фактическое количество отображаемых цифр показателя степени зависит от системы):

```
Введите значение с плавающей запятой: 64.25
Запись с фиксированной запятой: 64.250000
Экспоненциальная форма записи: 6.425000e+01
Двоично-экспоненциальное представление: 0x1.01p+6
```

5. В году содержится примерно 3.156×10^7 секунд. Напишите программу, которая предлагает ввести возраст в годах, а затем выводит на экран эквивалентное значение в секундах.
6. Масса одной молекулы воды приблизительно составляет 3.0×10^{-23} грамм. Кварта воды весит примерно 950 грамм. Напишите программу, которая предлагает ввести значение объема воды в квартах и отображает количество молекул воды в этом объеме.
7. В дюйме имеется 2.54 сантиметра. Напишите программу, которая предлагает ввести рост в дюймах, после чего выводит на экран этот рост в сантиметрах. Либо, если вам так больше нравится, программа может запрашивать рост в сантиметрах и переводить его в дюймы.
8. В американской системе единиц измерений объема пинта равна 2 чашкам, чашка — 8 унциям, унция — 2 столовым ложкам, а столовая ложка — 3 чайным ложкам. Напишите программу, которая предлагает ввести объем в чашках и отображает эквивалентные значения в пинтах, унциях, столовых ложках и чайных ложках. Почему для этой программы тип с плавающей запятой подходит больше, чем целочисленный?

4

Символьные строки и форматированный ввод-вывод

В ЭТОЙ ГЛАВЕ...

- Функция: strlen ()
- Ключевое слово: const
- Символьные строки
- Создание и храпение символьных строк
- Использование функций printf () и scant () для чтения и отображения символьных строк
- Использование функции strlen () для измерения длины строки
- Использование директивы #define препроцессора С и модификатора const стандарта ANSI С для создания символических констант

В этой главе основное внимание сосредоточено на вводе и выводе. После изучения всего предлагаемого здесь материала вы сможете придать своим программам индивидуальность, сделав их интерактивными и использующими символьные строки. Кроме того, более подробно рассматриваются две удобные функции ввода-вывода — printf() и scanf(). Эти функции являются программными инструментами для взаимодействия с пользователями и форматирования выходных данных в соответствие с конкретными потребностями и предпочтениями. Наконец, вы вкратце ознакомитесь с таким важным средством языка С, как препроцессор, и узнаете, каким образом определять и применять символические константы.

Вводная программа

К этому времени вы, вероятно, уже привыкли, что в начале каждой главы следует ожидать очередной простой учебной программы. Именно такая программа, реализующая диалог с пользователем, представлена в листинге 4.1. Чтобы внести некоторое разнообразие, в ней используется новый стиль комментариев.

Листинг 4.1. Программа talkback.c

```
// talkback.c -- любопытная информативная программа
#include <stdio.h>
#include <string.h>      // для прототипа функции strlen()
#define DENSITY 62.4     // удельная масса человека в фунтах на кубический фут
int main()
{
    float weight, volume;
    int size, letters;
    char name[40];       // name представляет собой массив из 40 символов

    printf("Здравствуйте! Как вас зовут?\n");
    scanf("%s", name);
    printf("%s, сколько вы весите в фунтах?\n", name);
    scanf("%f", &weight);
    size = sizeof name;
    letters = strlen(name);
    volume = weight / DENSITY;
    printf("Хорошо, %s, ваш объем составляет %2.2f кубических футов.\n",
           name, volume);
    printf("К тому же ваше имя состоит из %d букв,\n",
           letters);
    printf("и мы располагаем 40 байтами для его сохранения.\n", size);

    return 0;
}
```

Запустив на выполнение программу talkback.c, получаем следующий результат:

```
Здравствуйте! Как вас зовут?
Кристина
Кристина, сколько вы весите в фунтах?
154
Хорошо, Кристина, ваш объем составляет 2.47 кубических футов.
К тому же ваше имя состоит из 8 букв,
и мы располагаем 40 байтами для его сохранения.
```

Эта программа отличается следующими новыми особенностями.

- В ней применяется *массив* для хранения *символьной строки*. Имя пользователя считывается в массив, в этом случае представляющий собой набор из 40 последовательных байтов памяти, каждый из которых способен хранить значение одного символа.
- В рассматриваемой программе для обработки ввода и вывода строки используется *спецификатор преобразования* `%s`. Обратите внимание, что с переменной `name`, в отличие от `weight`, префикс `&` не указывается, когда она применяется в вызове функции `scanf()`. (Позже вы увидите, что как `&weight`, так и `name` являются адресами.)
- В программе используется препроцессор C для определения символической константы `DENSITY`, представляющей значение `62.4`.
- В рассматриваемой программе для выяснения длины строки применяется функция `strlen()`.

Подход к вводу-выводу, принятый в C, может показаться несколько усложненным по сравнению, скажем, с языком BASIC. Однако благодаря этой сложности достигается более точный контроль над вводом-выводом и высокая эффективность программ. Как только вы привыкнете к нему, он покажется удивительно простым.

Давайте ознакомимся с этими новыми идеями.

Введение в символьные строки

Символьная строка – это последовательность из одного или большего количества символов, например:

```
"Это длинная строка символов."
```

Двойные кавычки не являются частью строки. Они сообщают компилятору, что внутри них содержится строка, точно так же, как одиночные кавычки идентифицируют символ.

Массив типа char и нулевой символ

В языке C не существует какого-то специального типа для строковых переменных. Вместо этого для строк применяются массивы типа `char`. Символы в строке хранятся в смежных ячейках памяти, по одному символу на ячейку, а массив состоит из смежных ячеек памяти, так что строка размещается в массиве вполне естественным образом (рис. 4.1).

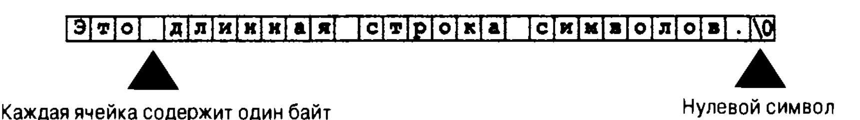


Рис. 4.1. Строка в массиве

На рис. 4.1 обратите внимание, что в последней позиции массива находится символ `\0`. Он представляет собой *нулевой символ*, который в языке C служит для пометки конца строки. Нулевой символ – это не цифра ноль, а непечатаемый символ, кодовое значение которого в кодировке ASCII (или эквивалентной) равно `0`. Строки в C всегда сохраняются с завершающим нулевым символом.

Присутствие нулевого символа означает, что массив должен иметь, по крайней мере на одну ячейку больше, чем количество символов, которые требуется сохранить. Таким образом, когда приведенная программа сообщает, что она располагает 40 байтами для строки, это означает, что она может хранить вплоть до 39 символов плюс нулевой символ.

Что же такое массив? Массив можно представить как несколько ячеек памяти, расположенных подряд. Если вы предпочитаете более формальный стиль, то массив – это упорядоченная последовательность элементов данных одного типа. В рассматриваемом примере создается массив из 40 ячеек памяти, или *элементов*, каждый из которых может хранить одно значение типа char, для чего используется следующее объявление:

```
char name[40];
```

Квадратные скобки после имени name идентифицируют его как массив. Число 40 внутри скобок указывает количество элементов в этом массиве. char идентифицирует тип каждого элемента (рис. 4.2).

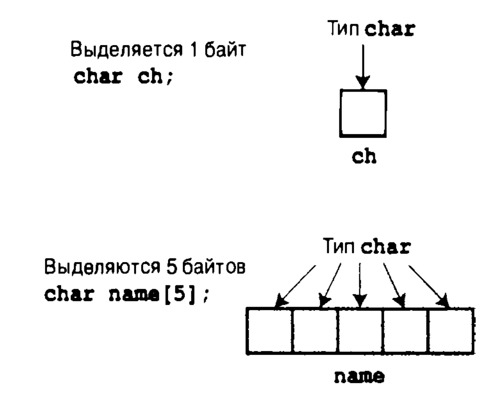


Рис. 4.2. Сравнение объявлений простой переменной и массива

Применение символьных строк начинает казаться излишне сложным. Вы должны создать массив, поместить внутрь него символы строки друг за другом, а еще не забыть добавить в конец массива символ \0. К счастью, компьютер может самостоятельно позаботиться о большинстве деталей.

Использование строк

С помощью программы в листинге 4.2 можно удостовериться, насколько просто в действительности работать со строками.

Листинг 4.2. Программа praise1.c

```
/* praise1.c -- использует различные представления строк */
#include <stdio.h>
#define PRAISE "Вы - выдающаяся личность."
int main(void)
{
   char name[40];
   printf("Как вас зовут? ");
   scanf("%s", name);
   printf("Здравствуйте, %s. %s\n", name, PRAISE);
   return 0;
}
```

Спецификатор %s сообщает функции printf() о необходимости вывода строки. Он встречается дважды, т.к. программа выводит две строки: одна хранится в массиве name, а другая представлена PRAISE. Выполнение программы praisel.c дает примерно такой результат:

```
Как вас зовут? Мария Иванова
Здравствуйте, Мария. Вы - выдающаяся личность.
```

Вам не придется самостоятельно помещать нулевой символ в массив name. Эту задачу решает функция scanf(), когда считывает входные данные. Точно так же нет необходимости во включении нулевого символа в *строковую символьную константу* PRAISE. Действия оператора #define мы рассмотрим позже, а пока просто запомните, что двойные кавычки, в которые заключается текст, следующий за PRAISE, идентифицируют данный текст как строку. Компилятор сам позаботится о добавлении нулевого символа.

Обратите внимание (и это важно) на то, что функция scanf() читает только имя Мария, а не имя и фамилию. После того, как функция scanf() начинает считывать входные данные, она останавливает чтение на первом встреченном *пробельном символе* (символе пробела, табуляции или новой строки). Таким образом, считывание для массива name прекращается, когда появляется символ пробела между словами "Мария" и "Иванова". В принципе функция scanf() применяется со спецификатором %s только для чтения одиночного слова, а не целой фразы, которая может находиться в строке. В языке C доступны другие функции ввода данных, такие как fgets(), поддерживающая общие строки. Эти функции подробно рассматриваться в последующих главах.

Различия между строками и символами

Строковая константа "x" — вовсе не то же самое, что и символьная константа 'x'. Одно из различий связано с тем, что 'x' имеет базовый тип (char), но "x" — это производный тип, представляющий собой массив значений char. Второе различие заключается в том, что "x" на самом деле состоит из двух символов — 'x' и '\0' (рис. 4.3).

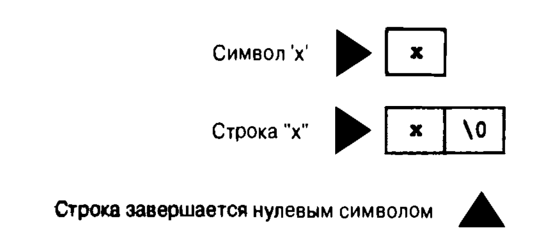


Рис. 4.3. Символ 'x' и строка "x"

Функция strlen()

В предыдущей главе кратко затрагивалась операция sizeof, которая предоставляет размер типа в байтах. Функция strlen() возвращает длину строки в символах. Поскольку для размещения одного символа требуется один байт, можно было бы предположить, что применительно к строке sizeof и strlen() дадут один и тот же результат, однако это не так. Добавьте в пример кода несколько строк, как показано в листинге 4.3, и вы поймете причину.

Листинг 4.3. Программа praise2.c

```
/* praise2.c */
// Если ваша реализация не распознает спецификатор %zd,
// попробуйте воспользоваться %u или %lu
#include <stdio.h>
#include <string.h>     /* предоставляет прототип strlenf() */
#define PRAISE "Вы - выдающаяся личность."
int main(void)
{
   char name[40];

   printf("Как вас зовут? ");
   scanf("%s", name);
   printf("Здравствуйте, %s. %s\n", name, PRAISE);
   printf("Ваше имя состоит из %zd символов и занимает %zd ячеек памяти.\n",
          strlen(name), sizeof name);
   printf("Хвалебная фраза содержит %zd символов",
          strlen(PRAISE));
   printf("и занимает %zd ячеек памяти.\n", sizeof PRAISE);

   return 0;
}
```

Если вы используете версию компилятора, не поддерживающую ANSI C, придется удалить следующую строку:

```
#include <string.h>
```

Заголовочный файл string.h содержит прототипы для нескольких функций обработки строк, включая strlen(). Более подробно этот заголовочный файл обсуждается в главе 11. (Кстати, в некоторых системах UNIX, разработанных до появления стандарта ANSI, вместо string.h применяется заголовочный файл strings.h, содержащий объявления строковых функций.)

В общем случае C разделяет библиотеку функций на семейства связанных функций и предоставляет отдельный заголовочный файл для каждого семейства. Например, функции printf() и scanf() принадлежат семейству стандартных функций ввода-вывода и имеют свой заголовочный файл stdio.h. Функция strlen() объединяет вокруг себя ряд других функций обработки строк, таких как функции для копирования и поиска в строках, и это семейство обслуживается заголовочным файлом string.h.

Следует отметить, что в листинге 4.3 длинные операторы printf() представлены с использованием двух методов. Первый метод предусматривает разнесение оператора printf() на две строки. (Вызов можно разделять в промежутках между аргументами, но не в середине строки; т.е. не между кавычками.) Второй метод предполагает применение для вывода одной строки двух операторов printf(). Символ новой строки (\n) присутствует только во втором операторе. После запуска программы возникает следующее взаимодействие с пользователем:

```
Как вас зовут? Васисуалий Лоханкин
Здравствуйте, Васисуалий. Вы - выдающаяся личность.
Ваше имя состоит из 10 букв и занимает 40 ячеек памяти.
Хвалебная фраза содержит 31 символов и занимает 32 ячеек памяти.
```

Давайте взглянем, что происходит. Массив name имеет 40 ячеек памяти, и именно об этом сообщает операция sizeof. Однако для размещения имени Васисуалий необходимы только первые 10 ячеек, и об этом информирует функция strlen().

Одиннадцатая ячейка в массиве name содержит нулевой символ, и его присутствие сообщает функции strlen(), когда она должна остановить подсчет. На рис. 4.4 эта концепция иллюстрируется на примере более короткой строки.

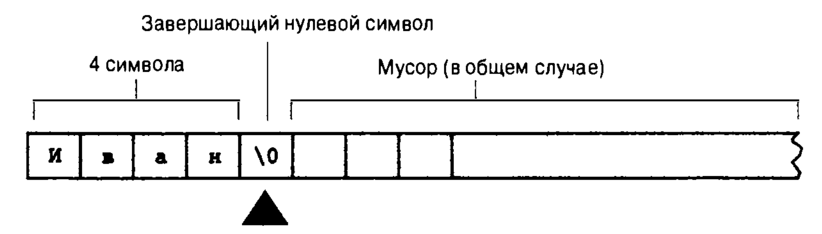


Рис. 4.4. Функции strlen () *известно, когда остановить подсчет символов*

Когда дело доходит до PRAISE, то обнаруживается, что strlen() снова возвращает точное количество символов в строке (включая пробелы и знаки препинания). Операция sizeof дает число, которое на единицу больше количества символов, т.к. она учитывает невидимый нулевой символ, обозначающий конец строки. В коде компьютеру не было указано, какой объем памяти нужно выделить для хранения фразы. Он должен самостоятельно подсчитать количество символов между двойными кавычками.

Как упоминалось в главе 3, стандарты C99 и C11 предполагают использование спецификатора %zd для типа, указываемого в операции sizeof. Это также относится и к типу, возвращаемому функцией strlen(). Для более ранних версий С необходимо знать действительный тип, возвращаемый операцией sizeof и функцией strlen(); обычно им будет unsigned int или unsigned long.

Еще один момент: в предыдущей главе операция sizeof применялась с круглыми скобками, но в этом примере их нет. Используете вы круглые скобки или нет, зависит от того, хотите вы получить размер типа или конкретной величины. Круглые скобки обязательны для типов, но необязательны для отдельных величин. Это значит, что вы будете применять sizeof(char) или sizeof(float), но также можете использовать sizeof name или sizeof 6.28. Тем не менее, в этих случаях также допускается указание круглых скобок, например, sizeof (6.28).

В последнем примере strlen() и sizeof применялись с довольно тривиальной целью удовлетворить потенциальное любопытство пользователя. Но в действительности функции strlen() и sizeof являются важными инструментами программирования. Например, как будет показано в главе 11, функция strlen() полезна во всех видах программ, работающих с символьными строками.

Давайте перейдем к рассмотрению оператора #define.

Константы и препроцессор С

Иногда в программе необходимо использовать константы. Например, длину окружности можно вычислить по формуле:

```
circumference = 3.14159 * diameter;
```

Здесь константа 3.14159 представляет общеизвестную константу π. Для применения константы просто введите ее действительное значение, как в приведенном примере. Однако существуют обоснованные причины, чтобы вместо значения использовать *символическую константу*. Это означает, что можно записать оператор, как показано ниже, и заставить компьютер позже подставить действительное значение:

```
circumference = pi * diameter;
```

Почему лучше применять символическую константу? Прежде всего, имя является более информативным, нежели число. Сравните следующие два оператора:

```
owed = 0.015 * housevalue;
owed = taxrate * housevalue;
```

В длинной программе понять второй оператор гораздо проще, чем первый.

Также предположим, что вы использовали константу в нескольких местах, и возникла необходимость изменить ее значение. В конце концов, даже налоговые ставки иногда меняются. В таком случае понадобится модифицировать только определение символической константы, а не искать и корректировать каждое вхождение числовой константы в программе.

А как установить символическую константу? Один из способов предусматривает объявление переменной и присваивание ей значения, которое равно желаемой константе. Можно было бы написать такой код:

```
float taxrate;
taxrate = 0.015;
```

Это предоставляет символическое имя, но `taxrate` является переменной, и в программе можно случайно изменить ее значение. К счастью, в языке С доступна пара более эффективных приемов.

Изначально лучшая идея предполагает применение препроцессора С. В главе 2 вы уже видели, как с помощью директивы `#include` препроцессора включать информацию из другого файла. Препроцессор также позволяет определять константы. Просто добавьте в начало файла, содержащего программу, строку следующего вида:

```
#define TAXRATE 0.015
```

После компиляции программы значение `0.015` будет подставлено повсюду, где использовалась константа `TAXRATE`. Это называется *подстановкой во время компиляции*. К моменту запуска программы все подстановки уже сделаны (рис. 4.5). Константы, определенные подобным образом, часто называются *символическими константами* или литералами.

Взгляните на формат. Сначала идет директива `#define`. За ней следует символическое имя (`TAXRATE`) для константы и затем ее значение (`0.015`). (Обратите внимание, что в этой конструкции отсутствует знак =.) Общая форма выглядит так:

```
#define ИМЯ значение
```

Здесь вы должны заменить конструкцию *ИМЯ* желаемым символическим именем, а конструкцию *значение* соответствующим значением. Точка с запятой в этом случае не указывается, т.к. это механизм замены, поддерживаемый препроцессором, а не оператор языка С. Почему имя `TAXRATE` записано прописными буквами? По сложившейся традиции имена констант в С представляются прописными буквами. Если где-то в недрах программы встречается имя подобного рода, то сразу становится ясно, что оно определяет константу, а не переменную. Представление имен констант прописными буквами является еще одним способом улучшения читабельности программ. Программы сохранят работоспособность и без представления констант прописными буквами, но разумнее взять этот прием на вооружение.

Другое менее распространенное соглашение по именованию констант предусматривает предварение имени префиксом `c_` или `k_` для указания на то, что оно представляет константу, в результате чего появляются имена, подобные `c_level` или `k_line`.

Имена, выбираемые для символических констант, должны удовлетворять тем же правилам, что и имена переменных.

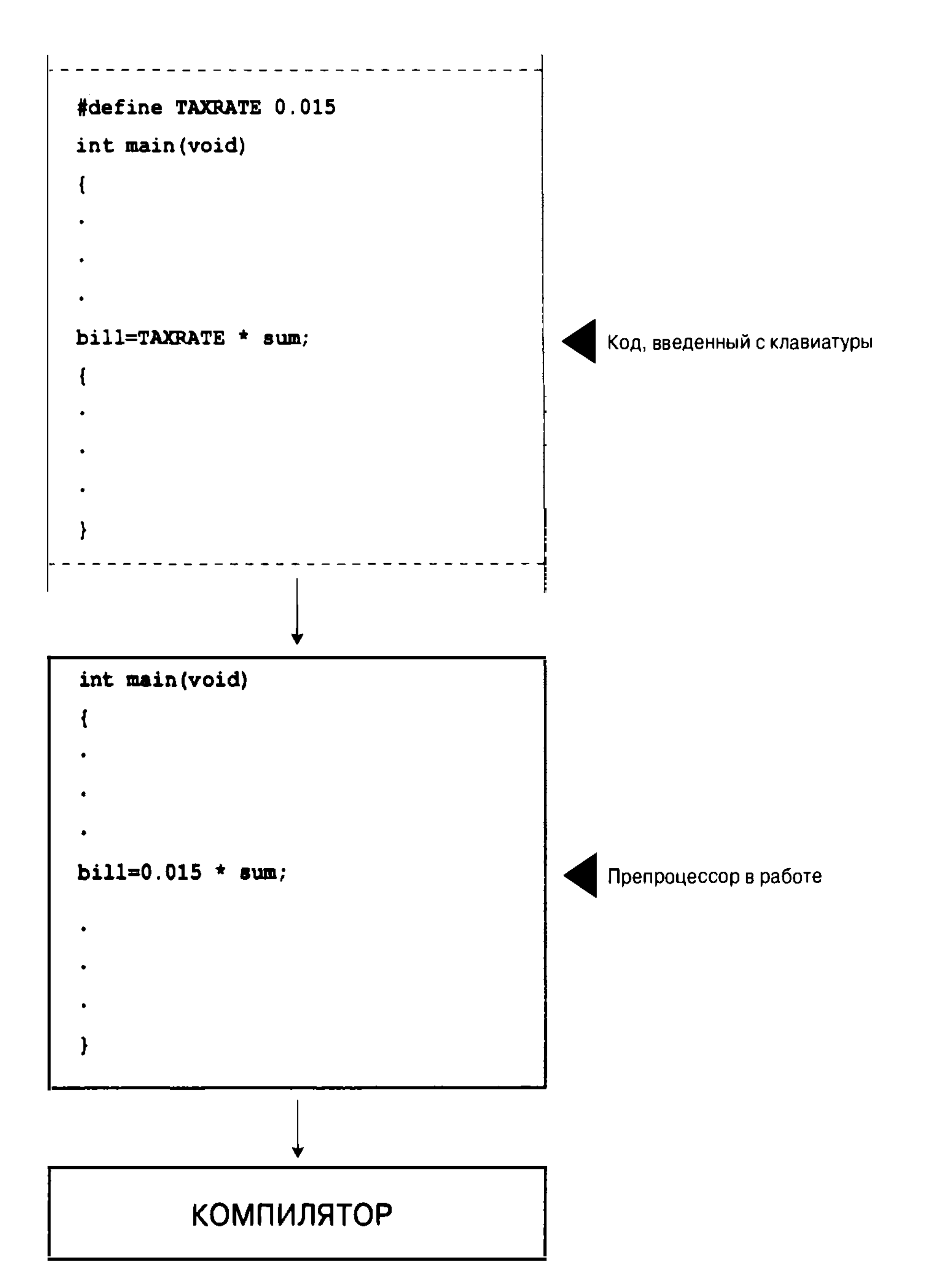


Рис. 4.5. Код, введенный с клавиатуры, и код, полученный в результате компиляции

Можно применять прописные и строчные буквы, цифры и символ подчеркивания. Первый символ не может быть цифрой. В листинге 4.4 приведен простой пример.

Листинг 4.4. Программа pizza.c

```
/* pizza.c -- использует константы, определенные применительно к пицце */
#include <stdio.h>
#define PI 3.14159
int main(void)
{
   float area, circum, radius;

   printf("Каков радиус вашей пиццы?\n");
   scanf("%f", &radius);
   area = PI * radius * radius;
   circum = 2.0 * PI *radius;
   printf("Основные параметры вашей пиццы:\n");
   printf("длина окружности = %1.2f, площадь = %1.2f\n", circum,
          area);

   return 0;
}
```

Спецификатор %1.2f в операторе printf() вызывает округление при выводе до двух десятичных позиций. Конечно, эта программа может не отражать все характеристики пиццы, представляющие для вас интерес, но она заполняет небольшую нишу в мире программ, имеющих отношение к пицце. Вот пример выполнения этой программы:

```
Каков радиус вашей пиццы?
6.0
Основные параметры вашей пиццы:
длина окружности = 37.70, площадь = 113.10
```

Директиву #define можно использовать также для объявления символьных и строковых констант. Достаточно указать одиночные кавычки для символьных и двойные кавычки для строковых констант. Ниже приведены допустимые объявления констант:

```
#define BEEP '\a'
#define TEE 'T'
#define ESC '\033'
#define OOPS "Теперь вы сделали это!"
```

Не забывайте, что любые данные, указанные за символическим именем, будут замещать его. Остерегайтесь распространенной ошибки:

```
/* следующее определение некорректно */
#define TOES = 20
```

Если поступить так, то константа TOES будет заменена последовательностью = 20, а не просто 20. В этом случае оператор следующего вида:

```
digits = fingers + TOES;
```

преобразуется в такое ошибочное представление:

```
digits = fingers + = 20;
```

Модификатор const

В стандарт C90 был добавлен второй способ создания символических констант, при котором с помощью ключевого слова const объявление для переменной преобразуется в объявление для константы:

```
const int MONTHS = 12;  // MONTHS является символической константой для 12
```

Такое объявление делает MONTHS значением только для чтения. Это означает, что вы можете отображать MONTHS на экране и применять его в вычислениях, но не модифицировать значение MONTHS. Новый подход более гибок, чем прием с директивой #define; он позволяет объявить тип и обеспечивает больший контроль над тем, в каких частях программы может использоваться константа. В главе 12 обсуждается этот и другие способы применения модификатора const.

На самом деле, как будет показано в главе 14, в языке С имеется еще и третий способ создания символических констант — использование enum.

Работа с символическими константами

Заголовочные файлы limits.h и float.h содержат подробную информацию об ограничениях размеров, соответственно, целочисленных типов и типов с плавающей запятой. В каждом файле определена последовательность символических констант, которые применяются к реализации. Например, файл limits.h содержит строки, подобные следующим:

```
#define INT_MAX    +32767
#define INT_MIN    -32768
```

Эти константы представляют наибольшее и наименьшее возможные значения для типа int. Если в вашей системе используется 32-битный тип int, то данный файл предоставит другие значения для таких символических констант. В файле limits.h определены минимальные и максимальные значения для всех целочисленных типов. После включения файла limits.h вы можете применять такой код:

```
printf("Максимальное значение типа int в этой системе составляет %d\n", INT_MAX);
```

Если в системе используется четырехбайтный тип int, то файл limits.h, который поступает с этой системой, предоставит определения для INT_MAX и INT_MIN, соответствующие пределам четырехбайтного типа int. В табл. 4.1 приведен список некоторых констант, находящихся в файле limits.h.

Таблица 4.1. Некоторые символические константы из файла limits.h

Символическая константа	Что представляет
CHAR_BIT	Количество битов в типе char
CHAR_MAX	Максимальное значение типа char
CHAR_MIN	Минимальное значение типа char
SCHAR_MAX	Максимальное значение типа signed char
SCHAR_MIN	Минимальное значение типа signed char
UCHAR_MAX	Максимальное значение типа unsigned char
SHRT_MAX	Максимальное значение типа short
SHRT_MIN	Минимальное значение типа short

Окончание табл. 4.1

Символическая константа	Что представляет
USHRT_MAX	Максимальное значение типа unsigned short
INT_MAX	Максимальное значение типа int
INT_MIN	Минимальное значение типа int
UINT_MAX	Максимальное значение типа unsigned int
LONG_MAX	Максимальное значение типа long
LONG_MIN	Минимальное значение типа long
ULONG_MAX	Максимальное значение типа unsigned long
LLONG_MAX	Максимальное значение типа long long
LLONG_MIN	Минимальное значение типа long long
ULLONG_MAX	Максимальное значение типа unsigned long long

Аналогично, в файле float.h определены такие константы, как FLT_DIG и DBL_DIG, которые представляют количество значащих цифр, поддерживаемое типами float и double. В табл. 4.2 перечислены некоторые константы, определенные в файле float.h. (Можете открыть в текстовом редакторе заголовочный файл float.h, доступный в вашей системе, и ознакомиться с его содержимым.) Здесь приводятся только данные для типа float. Эквивалентные константы определены для типов double и long double; в их именах вместо FLT применяются строки DBL и LDBL. (В табл. 4.2 предполагается, что в системе числа с плавающей запятой представлены степенями 2.)

Таблица 4.2. Некоторые символические константы из файла float.h

Символическая константа	Что представляет
FLT_MANT_DIG	Количество битов в мантиссе типа float
FLT_DIG	Минимальное количество значащих десятичных цифр для типа float
FLT_MIN_10_EXP	Минимальное значение отрицательного десятичного порядка для типа float с полным набором значащих цифр
FLT_MAX_10_EXP	Максимальное значение положительного десятичного порядка для типа float
FLT_MIN	Минимальное значение для положительного числа типа float, сохраняющего полную точность
FLT_MAX	Максимальное значение для положительного числа типа float
FLT_EPSILON	Разница между 1.00 и минимальным значением float, которое больше 1.00

В листинге 4.5 демонстрируется использование данных из float.h и limits.h. (Следует отметить, что компилятор, который не полностью поддерживает стандарт C99, может не принять идентификатор LONG_MIN.)

Листинг 4.5. Программа defines.c

```
// defines.c -- использует именованные константы из файла limit.h и тип float.
#include <stdio.h>
#include <limits.h>    // пределы для целых чисел
#include <float.h>     // пределы для чисел с плавающей запятой
int main(void)
{
    printf("Некоторые пределы чисел для данной системы:\n");
    printf("Наибольшее значение типа int: %d\n", INT_MAX);
    printf("Наименьшее значение типа long long: %lld\n", LLONG_MIN);
    printf("В данной системе один байт = %d битов.\n", CHAR_BIT);
    printf("Наибольшее значение типа double: %e\n", DBL_MAX);
    printf("Наименьшее нормализованное значение типа float: %e\n", FLT_MIN);
    printf("Точность значений типа float = %d знаков\n", FLT_DIG);
    printf("Разница между 1.00 и минимальным значением float, которое больше 1.00 = %e\n",
            FLT_EPSILON);

    return 0;
}
```

Ниже показан пример вывода:

```
Некоторые пределы чисел для данной системы:
Наибольшее значение типа int: 2147483647
Наименьшее значение типа long long: -9223372036854775808
В данной системе один байт = 8 битов.
Наибольшее значение типа double: 1,797693e+308
Наименьшее нормализованное значение типа float: 1,175494e-38
Точность значений типа float = 6 знаков
Разница между 1.00 и минимальным значением float, которое больше 1.00 =
1.192093e-07
```

Препроцессор C является полезным и удобным инструментом, так что применяйте его везде, где это возможно. Далее в книге вы увидите и другие случаи его использования.

Исследование и эксплуатация функций printf() и scanf()

Функции printf() и scanf() позволяют организовать взаимодействие с программой и называются *функциями ввода-вывода*. В языке C доступны и другие функции ввода-вывода, но printf() и scanf() являются наиболее универсальными. Исторически сложилось так, что они, как и все остальные функции в библиотеке C, не были частью определения языка. Первоначально язык C оставлял реализацию средств ввода-вывода разработчикам компиляторов; это делало возможным лучшее соответствие функций ввода-вывода конкретным машинам. В интересах совместимости различные реализации поставлялись со своими версиями функций scanf() и printf(). Тем не менее, между реализациями встречались некоторые расхождения. В C90 и C99 описаны стандартные версии этих функций, и именно их мы будем придерживаться.

Хотя printf() является функцией вывода, а scanf() — функцией ввода, обе они работают очень похожим образом, используя управляющую строку и список аргументов. Давайте рассмотрим по очереди printf() и scanf().

Функция printf()

Инструкции, которые вы даете функции printf(), запрашивая у нее вывод переменной, зависят от типа этой переменной. Например, ранее мы применяли форму записи %d при выводе целого числа и %c при выводе символа. Эти обозначения называются *спецификаторами преобразования*, поскольку они определяют, каким образом данные преобразуются в отображаемую форму. Мы приведем список спецификаторов преобразования, которые стандарт ANSI C предоставляет для функции printf(), и затем покажем, как использовать наиболее общие из них. В табл. 4.3 перечислены спецификаторы преобразования и показан вывод, к которому они приводят.

Таблица 4.3. Спецификаторы преобразования и результирующий вывод

Спецификатор преобразования	Описание вывода
%a	Число с плавающей запятой, шестнадцатеричные цифры и p-запись (C99/C11)
%A	Число с плавающей запятой, шестнадцатеричные цифры и P-запись (C99/C11)
%c	Одиночный символ
%d	Десятичное целое число со знаком
%e	Число с плавающей запятой, экспоненциальное представление
%E	Число с плавающей запятой, экспоненциальное представление
%f	Число с плавающей запятой, десятичное представление
%g	В зависимости от значения использует %f или %e. Спецификатор %e применяется, если показатель степени меньше -4 либо больше или равен указанной точности
%G	В зависимости от значения использует %f или %E. Спецификатор %E применяется, если показатель степени меньше -4 либо больше или равен указанной точности
%i	Десятичное целое число со знаком (то же, что и %d)
%o	Восьмеричное целое число без знака
%p	Указатель
%s	Символьная строка
%u	Десятичное целое число без знака
%x	Шестнадцатеричное целое число без знака, используются шестнадцатеричные цифры 0f
%X	Шестнадцатеричное целое число без знака, используются шестнадцатеричные цифры 0F
%%	Знак процента

Использование функции printf()

В листинге 4.6 представлена программа, в которой применяются некоторые спецификаторы преобразования.

Листинг 4.6. Программа printout.c

```
/* printout.c -- использует спецификаторы преобразования */
#include <stdio.h>
#define PI 3.141593
int main(void)
{
   int number = 7;
   float pies = 12.75;
   int cost = 7800;

   printf("%d участников соревнований съели %f пирожков с вишнями.\n", number,
          pies);
   printf("Значение pi равно %f.\n", PI);
   printf("До свидания! Ваше искусство слишком дорого обходится,\n");
   printf("%c%d\n", '$', 2 * cost);

   return 0;
}
```

Вывод программы выглядит вполне ожидаемо:

```
7 участников соревнований съели 12.750000 пирожков с вишнями.
Значение pi равно 3.141593.
До свидания! Ваше искусство слишком дорого обходится,
$15600
```

Формат использования функции `printf()` имеет вид:

```
printf(управляющая-строка, элемент1, элемент2, ...);
```

Здесь *элемент1*, *элемент2* и т.д. – это элементы, которые нужно вывести. Ими могут быть переменные, константы или даже выражения, которые вычисляются до того, как значение будет выведено. Далее, *управляющая-строка* представляет собой символьную строку, описывающую способ вывода элементов. Как упоминалось в главе 3, управляющая строка должна содержать спецификатор преобразования для каждого выводимого элемента. Например, рассмотрим следующий оператор:

```
printf("%d участников соревнований съели %f пирожков с вишнями.\n", number,
       pies);
```

В этом операторе *управляющая-строка* – это фраза, заключенная в двойные кавычки. Она содержит два спецификатора преобразования, соответствующие `number` и `pies` – двум выводимым элементам. На рис. 4.6 показан другой пример применения оператора `printf()`.

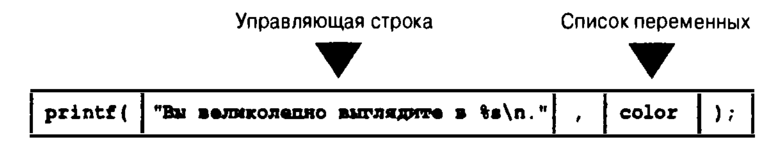


Рис. 4.6. Аргументы функции `printf()`

Вот еще одна строка из примера:

```
printf("Значение pi равно %f.\n", PI);
```

На этот раз список элементов состоит только из одного элемента – символической константы `PI`.

Как можно видеть на рис. 4.7, управляющая строка содержит два разных вида информации:

- символы, которые в действительности выводятся;
- спецификаторы преобразования.

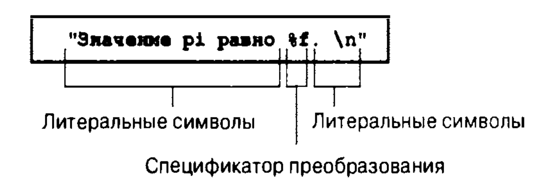


Рис. 4.7. Структура управляющей строки

Внимание!

Не забудьте предусмотреть по одному спецификатору преобразования для каждого элемента, следующего за управляющей строкой. Если вы не удовлетворите это основное требование, возникнут проблемы. Никогда не поступайте так:

```
printf("Выпало %d очков из %d.\n", score1);
```

Здесь отсутствует значение для второго спецификатора %d. Результат такой небрежности зависит от системы, но в лучшем случае на экране отобразятся бессмысленные символы.

Если вы хотите вывести только фразу, то не нуждаетесь в каких-либо спецификаторах преобразования. Чтобы вывести только данные, вы можете обойтись традиционным вызовом. Оба следующих оператора из листинга 4.6 вполне приемлемы:

```
printf("До свидания! Ваше искусство слишком дорого мне обходится,\n");
printf("%c%d\n", '$', 2 * cost);
```

Обратите внимание, что во втором операторе первый элемент в списке для вывода представляет собой символьную константу, а не переменную, тогда как второй элемент является результатом умножения. Это иллюстрирует тот факт, что функция `printf()` работает со значениями, будь они переменными, константами или выражениями.

Поскольку функция `printf()` использует символ % для идентификации спецификаторов преобразования, то возникает небольшая проблема, когда требуется вывести сам символ %. Если просто указать одиночный знак %, компилятор посчитает, что вы некорректно задали спецификатор преобразования. Выход из этой ситуации прост: достаточно указать два символа %:

```
pc = 2*6;
printf("Только %d%% припасов Мэри были пригодными в пищу.\n", pc);
```

В результате выполнения этого фрагмента программы получим следующий результат:

```
Только 12% припасов Мэри были пригодными в пищу.
```

Модификаторы спецификаторов преобразования для функции `printf()`

Базовый спецификатор преобразования можно изменять, вставляя модификаторы между знаком % и символом, который определяет преобразование.

В табл. 4.4 и 4.5 перечислены символы, которые можно здесь размещать. При указании более одного модификатора они должны располагаться в том же порядке, в каком они представлены в табл. 4.4. Не все возможные комбинации допустимы. В таблице отражены дополнения стандарта C99; ваша реализация может не поддерживать все показанные варианты.

Таблица 4.4. Модификаторы функции printf()

Модификатор	Описание
флаг	Пять допустимых флагов (-, +, пробел, # и 0) описаны в табл. 4.5. Можно указывать ноль или больше флагов. Пример: "%-10d"
цифра (цифры)	Минимальная ширина поля. Если выводимое число или строка не умещаются в это поле, будет использоваться поле большей ширины. Пример: "%4d"
.цифра (.цифры)	Точность. Для преобразований %e, %E и %f указывается количество цифр, которые будут выведены справа от десятичной точки. Для преобразований %g и %G задается максимальное количество значащих цифр. Для преобразований %s определяется максимальное количество символов, которое может быть выведено. Для целочисленных преобразований указывается минимальное количество отображаемых цифр; при необходимости для соответствия с этим минимумом применяются ведущие нули. Использование только точки (.) предполагает, что далее следует ноль, т.е %.f — то же самое, что и %.0f. Пример: "%5.2f" выводит значение типа float в поле шириной пять символов и двумя цифрами после десятичной точки
h	Используется со спецификатором целочисленного преобразования для отображения значений типа short int или unsigned short int. Примеры: "%hu", "%hx" и "%6.4hd".
hh	Используется со спецификатором целочисленного преобразования для отображения значений типа signed char или unsigned char. Примеры: "%hhu", "%hhx" и "%6.4hhd"
j	Используется со спецификатором целочисленного преобразования для отображения значений intmax_t или uintmax_t; эти типы определены в stdint.h. Примеры: "%jd" и "%8jX"
l	Используется со спецификатором целочисленного преобразования для отображения значений типа long int или unsigned long int. Примеры: "%ld" и "%8lu"
ll	Используется со спецификатором целочисленного преобразования для отображения значений типа long long int или unsigned long long int (стандарт C99). Примеры: "%lld" и "%8llu"
L	Используется со спецификатором преобразования значений с плавающей запятой для отображения значений типа long double. Примеры: "%Lf" и "%10.4Le"
t	Используется со спецификатором целочисленного преобразования для отображения значений ptrdiff_t. Этот тип соответствует разницы между двумя указателями (стандарт C99). Примеры: "%td" и "%12ti"
z	Используется со спецификатором целочисленного преобразования для отображения значений size_t. Этот тип возвращается операцией sizeof (стандарт C99). Примеры: "%zd" и "%12zx"

НА ЗАМЕТКУ! Переносимость типов

Вспомните, что операция sizeof возвращает размер типа или значения в байтах. Это значение должно быть какой-либо формой целого числа, но стандарт допускает только целое значение без знака. Следовательно, им может быть unsigned int, unsigned long или даже unsigned long long. Таким образом, в случае применения функции printf() для отображения выражения sizeof можно было бы использовать спецификатор %u в одной системе, %lu — в другой и %llu — в третьей. Это значит, что нужно выяснить правильное применение в конкретной системе, и что в случае переноса в другую систему может потребоваться изменить программу.

Итак, помимо всего прочего, язык C предоставляет поддержку для обеспечения более высокой переносимости типов. Во-первых, заголовочный файл stddef.h (включаемый в результате включения заголовочного файла stdio.h) определяет, что типом size_t будет тип, используемый в системе для возвращаемого значения операции sizeof. Этот тип называется основополагающим типом. Во-вторых, в функции printf() применяется модификатор z для указания соответствующего типа при выводе. Аналогично в языке C определен тип ptrdiff_t и модификатор t для указания основополагающего целочисленного типа со знаком, используемого системой для представления разницы между двумя адресами.

НА ЗАМЕТКУ! Преобразование аргументов типа float

Существуют спецификаторы преобразования для вывода типов double и long double. В то же время такой спецификатор для типа float отсутствует. Причина в том, что в классическом языке K&R C значения типа float автоматически преобразовывались в тип double перед использованием в выражении или до передачи в качестве аргумента. В общем случае в стандарте ANSI C (или последующих реализациях) не предусматривается автоматическое преобразование float в double. Однако для того, чтобы обеспечить правильную работу огромного количества существующих программ, которые разрабатывались с расчетом на то, что аргументы типа float преобразуются в double, все аргументы float для функции printf() — и других функций C, не использующих явные прототипы — автоматически преобразуются в тип double. Поэтому ни в K&R C, ни в ANSI C специальный спецификатор преобразования для отображения типа float не требуется.

Таблица 4.5. Флаги функции printf()

Флаг	Описание
-	Элемент выравнивается влево, т.е. содержимое будет выведено, начиная от левой границы поля. Пример: "%-20s"
+	Значения со знаком выводятся со знаком +, если они положительные, и со знаком -, если отрицательные. Пример: "%+6.2f"
пробел	Значения со знаком выводятся с ведущим пробелом (но без знака), если они положительны, и со знаком -, если отрицательные. Флаг + переопределяет действие пробела. Пример: "% 6.2f"
#	Использует альтернативную форму для спецификаторов преобразования. Выводит ведущий 0 для формы %o и ведущий 0x или 0X для форм %x и %X. Для всех форм с плавающей запятой флаг # гарантирует, что символ десятичной точки будет выведен, даже если за ним не следуют цифры. Для форм %g и %G это предотвращает удаление завершающих нулей. Примеры: "%#o", "%#8.0f" и "%+#10.3E"
0	Для числовых форм этот флаг приводит к заполнению пустых позиций поля ведущими нулями, а не пробелами. Данный флаг игнорируется, если присутствует флаг - или если для целочисленной формы указана точность. Примеры: "%010d" и "%08.3f".

Примеры использования модификаторов и флагов

Давайте посмотрим на описанные выше модификаторы в действии. Мы начнем с оценки влияния модификатора, устанавливающего ширину поля, на вывод целого числа. Рассмотрим программу, показанную в листинге 4.7.

Листинг 4.7. Программа width.c

```
/* width.c -- поля разной ширины */
#include <stdio.h>
#define PAGES 959
int main(void)
{
   printf("*%d*\n", PAGES);
   printf("*%2d*\n", PAGES);
   printf("*%10d*\n", PAGES);
   printf("*%-10d*\n", PAGES);

   return 0;
}
```

Программа из листинга 4.7 выводит одно и то же число четыре раза, применяя четыре разных спецификатора преобразования. Звездочка (*) служит для обозначения начала и конца каждого поля. Вывод выглядит следующим образом:

```
*959*
*959*
*       959*
*959       *
```

Первым спецификатором преобразования является %d без модификаторов. Он производит поле с шириной, которую имеет выводимое целое число. Этот вариант принят по умолчанию, т.е. если не предоставлены дальнейшие инструкции, то число будет выведено именно в таком виде. Второй спецификатор преобразования – %2d. Он устанавливает ширину поля равной 2, но поскольку в рассматриваемом примере целое число имеет три значащих цифры, поле автоматически расширяется, чтобы уместить это число. Следующий спецификатор преобразования – %10d. Он генерирует поле шириной 10 символов, при этом в итоге получаются семь пробелов и три цифры между звездочками, а число смещено к правой границе поля. Последним спецификатором является %-10d. Он также производит поле шириной 10 символов, а знак - означает, что число начинается с левого края, как и было заявлено. Привыкнув к этой системе, вы убедитесь, что она проста в применении и обеспечивает высокий контроль над внешним видом вывода. Попробуйте изменить значение PAGES, чтобы посмотреть, как выводятся числа с различным количеством цифр.

Теперь рассмотрим форматы чисел с плавающей запятой. Введите, скомпилируйте и запустите программу, показанную в листинге 4.8.

Листинг 4.8. Программа floats.c

```
// Программа floats.c -- некоторые комбинации для типов с плавающей запятой
#include <stdio.h>

int main(void)
{
   const double RENT = 3852.99; // константа, объявленная посредством const

   printf("*%f*\n", RENT);
   printf("*%e*\n", RENT);
```

```
    printf("*%4.2f*\n", RENT);
    printf("*%3.1f*\n", RENT);
    printf("*%10.3f*\n", RENT);
    printf("*%10.3E*\n", RENT);
    printf("*%+4.2f*\n", RENT);
    printf("*%010.2f*\n", RENT);

    return 0;
}
```

На этот раз для создания символической константы в программе используется ключевое слово const. Вывод имеет следующий вид:

```
*3852.990000*
*3.852990e+03*
*3852.99*
*3853.0*
*  3852.990*
* 3.853E+03*
*+3852.99*
*0003852.99*
```

Пример начинается с версии, применяемой по умолчанию — %f. В этом случае задействованы два стандартных параметра: ширина поля и количество цифр справа от десятичной точки. Количество цифр по умолчанию равно шести, а ширина поля должна быть такой, чтобы уместить число.

Затем используется еще одна версия спецификатора, принятая по умолчанию — %e. Она выводит одну цифру слева от десятичной точки и резервирует шесть позиций справа от нее. Получается довольно много цифр. Чтобы исправить это положение, нужно указать количество десятичных позиций справа от десятичной точки, и следующие четыре примера служат иллюстрацией такого решения. Обратите внимание на то, что в четвертом и шестом примере при выводе происходит округление. Вдобавок в шестом примере вместо спецификатора e применяется E.

Наконец, флаг + приводит к выводу результата с его алгебраическим знаком, которым в данном случае является "плюс", а флаг 0 обеспечивает дополнение до полной ширины поля ведущими нулями. Следует отметить, что в спецификаторе %010.2f первый 0 — это флаг, а остальные цифры до десятичной точки (10) указывают ширину поля.

Можете модифицировать значение RENT, чтобы посмотреть, как выводятся значения разнообразной длины. Программа в листинге 4.9 демонстрирует еще несколько возможных комбинаций.

Листинг 4.9. Программа flags.c

```
/* flags.c -- иллюстрирует применение некоторых флагов форматирования */
#include <stdio.h>
int main(void)
{
    printf("%x %X %#x\n", 31, 31, 31);
    printf("**%d**% d**% d**\n", 42, 42, -42);
    printf("**%5d**%5.3d**%05d**%05.3d**\n", 6, 6, 6, 6);

    return 0;
}
```

Вывод программы показан ниже:

```
1f 1F 0x1f
**42** 42**-42**
**    6** 006**00006**  006**
```

Первым делом отметим, что 1f — это шестнадцатеричный эквивалент десятичного числа 31. Спецификатор x выдает результат 1f, а спецификатор X — 1F. Использование флага # обеспечивает вывод ведущих символов 0x.

Вторая строка вывода иллюстрирует, что применение пробела в спецификаторе приводит к появлению ведущего пробела для положительных, но не для отрицательных значений. Это позволяет получать симпатичный вывод, т.к. положительные и отрицательные значения с одинаковым количеством значащих цифр выводятся в полях одинаковой ширины.

В третьей строке показано, что использование спецификатора точности (%5.3d) с целочисленной формой дополняет число ведущими нулями до получения минимального количества цифр (трех в данном случае). Однако применение флага 0 приводит к дополнению представления числа ведущими нулями, которых достаточно для заполнения всей ширины поля. Наконец, при одновременном указании флага 0 и спецификатора точности флаг 0 игнорируется.

Теперь исследуем некоторые варианты со строкой. Рассмотрим программу в листинге 4.10.

Листинг 4.10. Программа stringf.c

```
/* stringf.c — форматирование строк */
#include <stdio.h>
#define BLURB "Authentic imitation!"
int main(void)
{
   printf("[%2s]\n", BLURB);
   printf("[%24s]\n", BLURB);
   printf("[%24.5s]\n", BLURB);
   printf("[%-24.5s]\n", BLURB);

   return 0;
}
```

В результате выполнения программы получается следующий вывод:

```
[Authentic imitation!]
[    Authentic imitation!]
[                   Authe]
[Authe                   ]
```

Обратите внимание, что спецификатор %2s расширяет поле настолько, чтобы уместить все символы строки. Кроме того, спецификатор точности ограничивает количество выводимых символов. Конструкция .5 в спецификаторе формата сообщает функции printf() о том, что нужно вывести только пять символов. Опять-таки, модификатор - выравнивает текст по левому краю.

Использование полученных знаний на практике

Итак, вы ознакомились с несколькими примерами. Как должен выглядеть оператор для вывода текста в следующей форме:

```
Семья NAME может стать богаче на $XXX.XX!
```

Здесь NAME и XXX.XX представляют значения, которые будут предоставляться в программе переменными, скажем, name[40] и cash.

Одно из возможных решений выглядит так:

```
printf("Семья %s может стать богаче на $%.2f!\n", name, cash);
```

Что преобразует спецификатор преобразования?

Теперь более подробно рассмотрим, что именно преобразует спецификатор преобразования. Он преобразует значение, хранящееся в памяти компьютера в двоичном формате, в последовательность символов (строку) с целью отображения. Например, число 76 может быть представлено в памяти компьютера в двоичном виде как 01001100. Спецификатор преобразования %d превращает его в символы 7 и 6, отображая 76. Преобразование %x превращает это же двоичное значение (01001100) в шестнадцатеричное представление 4c, а спецификатор %c преобразует его в символьное представление L.

Термин *преобразование*, возможно, в чем-то неточен, т.к. можно предположить, что исходное значение заменяется преобразованным. Спецификаторы преобразования по существу являются спецификаторами трансляции; к примеру, %d означает "транслировать заданное значение в десятичное целочисленное текстовое представление и затем вывести его".

Несовпадающие преобразования

Естественно, спецификатор преобразования должен соответствовать типу выводимого значения. Часто вам доступно несколько вариантов. Например, для вывода значения типа int можно применять спецификатор %d, %x или %o. Все эти спецификаторы предполагают, что вы выводите значение типа int; они просто предоставляют различные представления этого значения. Аналогично, спецификаторы %f, %e или %g можно использовать для представления типа double.

Что произойдет, если спецификатор преобразования не соответствует типу? В предыдущей главе вы уже видели, что несоответствия могут вызвать проблемы. Это очень важный аспект, который следует иметь в виду, так что в листинге 4.11 приведено еще несколько примеров несоответствия при работе с семейством целочисленных типов.

Листинг 4.11. Программа intconv.c

```
/* intconv.c -- несоответствия при преобразовании целочисленных типов */
#include <stdio.h>
#define PAGES 336
#define WORDS 65618
int main(void)
{
    short num = PAGES;
    short mnum = -PAGES;

    printf("num как тип short и тип unsigned short: %hd %hu\n", num,
          num);
    printf("-num как тип short и тип unsigned short: %hd %hu\n", mnum,
          mnum);
    printf("num как тип int и тип char: %d %c\n", num, num);
    printf("WORDS как тип int, short и char: %d %hd %c \n",
          WORDS, WORDS, WORDS);

    return 0;
}
```

В нашей системе были получены следующие результаты:

```
num как тип short и тип unsigned short: 336 336
-num как тип short и тип unsigned short: -336 65200
num как тип int и тип char: 336 P
WORDS тип int, short и char: 65618 82 R
```

Взглянув на первую строку, вы можете заметить, что спецификаторы %hd и %hu выдают 336 в качестве вывода для переменной num; тут нет никаких проблем. Однако во второй строке версия %u (без знака) для mnum выглядит как 65200, а не как ожидаемое значение 336; это вытекает из способа представления значений типа short int со знаком в нашей системе. Во-первых, они имеют размер 2 байта. Во-вторых, для представления целых чисел со знаком система использует метод, называемый *поразрядным дополнением до двойки*. При таком методе числа от 0 до 32767 представляют сами себя, а числа от 32768 до 65535 представляют отрицательные числа, причем 65535 соответствует -1, 65534 — -2 и т.д. Следовательно, -336 представлено как 65536 - 336, или 65200. Таким образом, число 65200 представляет -336, когда интерпретируется как int со знаком, и 65200, когда интерпретируется как int без знака. Поэтому будьте осторожны! Одно число может интерпретироваться как два разных значения. Описанный метод представления отрицательных целых чисел применяется не во всех системах. Тем не менее, мораль этого примера: не рассчитывайте на то, что преобразование %u просто отбросит знак числа.

Третья строка демонстрирует, что происходит при попытке преобразования в символ значения, которое больше 255. В нашей системе тип short int занимает 2 байта, а тип char — 1 байт. Когда функция printf() выводит 336 с использованием спецификатора %c, она просматривает только один байт из двух, задействованных для хранения 336. Такое усечение (рис. 4.8) равнозначно делению целого числа на 256 с сохранением только остатка. В этом случае остаток равен 80, что представляет собой ASCII-значение символа *P*. Формально можно сказать, что число интерпретируется как результат деления *по модулю 256*, что означает использование остатка от деления числа на 256.

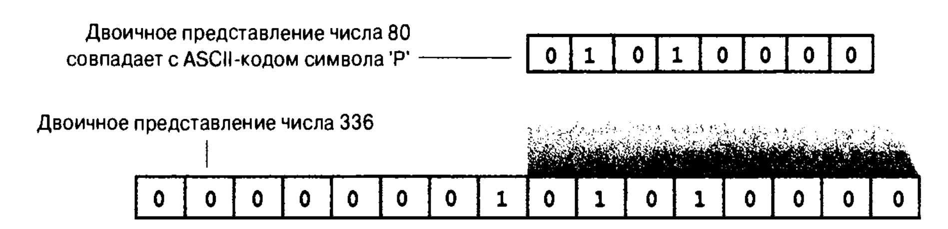


Рис. 4.8. Интерпретация числа 336 как символа

В заключение мы попытались вывести в своей системе целое число (65618), превышающее максимально допустимое значение типа short int (32767). И снова компьютер применил деление по модулю. Число 65618 в силу своего размера хранится в нашей системе как 4-байтовое значение int. Когда оно выводится с применением спецификатора %hd, функция printf() использует только последние 2 байта, которые равносильны остатку от деления на 65536. В этом случае остаток равен 82. Остаток, находящийся между 32767 и 65536, с учетом способа хранения отрицательных чисел выводился бы как отрицательное число. В системах с другими размерами целых чисел общее поведение было бы таким же, но с другими числовыми значениями.

Когда вы начнете смешивать целочисленные типы и типы с плавающей запятой, результаты станут еще более причудливыми. Для примера рассмотрим программу, приведенную в листинге 4.12.

Листинг 4.12. Программа floatcnv.c

```
/* floatcnv.c -- несогласованные преобразования с плавающей запятой */
#include <stdio.h>
int main(void)
{
   float n1 = 3.0;
   double n2 = 3.0;
   long n3 = 2000000000;
   long n4 = 1234567890;

   printf("%.1e %.1e %.1e %.1e\n", n1, n2, n3, n4);
   printf("%ld %ld\n", n3, n4);
   printf("%ld %ld %ld %ld\n", n1, n2, n3, n4);

   return 0;
}
```

В нашей системе код из листинга 4.12 сгенерировал следующий вывод:

```
3.0e+00 3.0e+00 3.1e+46 1.7e+266
2000000000 1234567890
0 1074266112 0 1074266112
```

Первая строка вывода показывает, что применение спецификатора %e не вызывает преобразование целого числа в число с плавающей запятой. Давайте посмотрим, что происходит при попытке вывода переменной n3 (типа long) с использованием спецификатора %e. Во-первых, спецификатор %e заставляет функцию printf() ожидать значение типа double, которое в нашей системе является 8-байтовым. Когда функция printf() исследует переменную n3, представленную в нашей системе 4-байтовым значением, она просматривает также смежные 4 байта памяти. Таким образом, функция анализирует 8-байтовый блок, в котором содержится действительное значение n3. Во-вторых, она интерпретирует биты этого блока как число с плавающей запятой. Например, некоторые биты, будут трактоваться в качестве показателя степени. Поэтому, даже если бы значение n3 содержало правильное количество битов, для спецификаторов %e и %ld они бы интерпретировались по-разному. Конечный результат оказывается бессмысленным.

Первая строка вывода также иллюстрирует то, что упоминалось ранее – при передаче в виде аргумента функции printf() значение float преобразуется в тип double. В данной системе тип float занимает 4 байта, но переменная n1 была расширена до 8 байтов, чтобы функция printf() смогла корректно отобразить ее значение.

Вторая строка вывода показывает, что функция printf() может правильно выводить значения n3 и n4, если указан корректный спецификатор.

Третья строка вывода демонстрирует, что даже правильный спецификатор может приводить к ложным результатам, если оператор printf() содержит несоответствия где-то в другом месте. Как и можно было ожидать, попытка вывода значения с плавающей запятой с применением спецификатора %ld оказывается неудачной, однако в данном случае неудачу терпит и попытка вывода значения типа long с использованием спецификатора %ld! Проблема кроется в способе передачи информации функции. Точные детали отказа при выводе зависят от реализации, но во врезке “Передача аргументов” обсуждается поведение в типичной системе.

Передача аргументов

Механизм передачи аргументов зависит от реализации. Вот как передача аргументов происходит в нашей системе. Вызов функции выглядит следующим образом:

```
printf("%ld %ld %ld %ld\n", n1, n2, n3, n4);
```

Этот вызов сообщает компьютеру о том, что ему передаются значения переменных n1, n2, n3 и n4. Ниже описан один из распространенных способов обработки этой ситуации. Программа помещает значения в область памяти, которая называется *стеком*. Когда компьютер помещает эти значения в стек, он руководствуется типами переменных, а не спецификаторами преобразования. Следовательно, для n1 он выделяет в стеке 8 байтов (float преобразуется в double). Подобным же образом для переменной n2 отводится еще 8 байтов, после чего по 4 байта выделяется для переменных n3 и n4. Затем управление передается функции printf(). Эта функция читает значения из стека, но делает это согласно спецификаторам преобразования. Спецификатор %ld указывает, что функция printf() должна прочитать 4 байта, поэтому она считывает первые 4 байта в стеке в качестве своего первого значения. Прочитанные 4 байта представляют собой первую половину n1, которая интерпретируется как целочисленное значение long. Следующий спецификатор %ld обеспечивает чтение еще 4 байтов; это вторая половина n1, и она интерпретируется как второе целочисленное значение long (рис. 4.9). Аналогично третий и четвертый спецификаторы %ld приводят к чтению первой и второй половины n2 с последующей их интерпретацией в качестве еще двух целочисленных значений long, так что, хотя для переменных n3 и n4 указаны корректные спецификаторы, функция printf() читает не те байты, которые нужны.

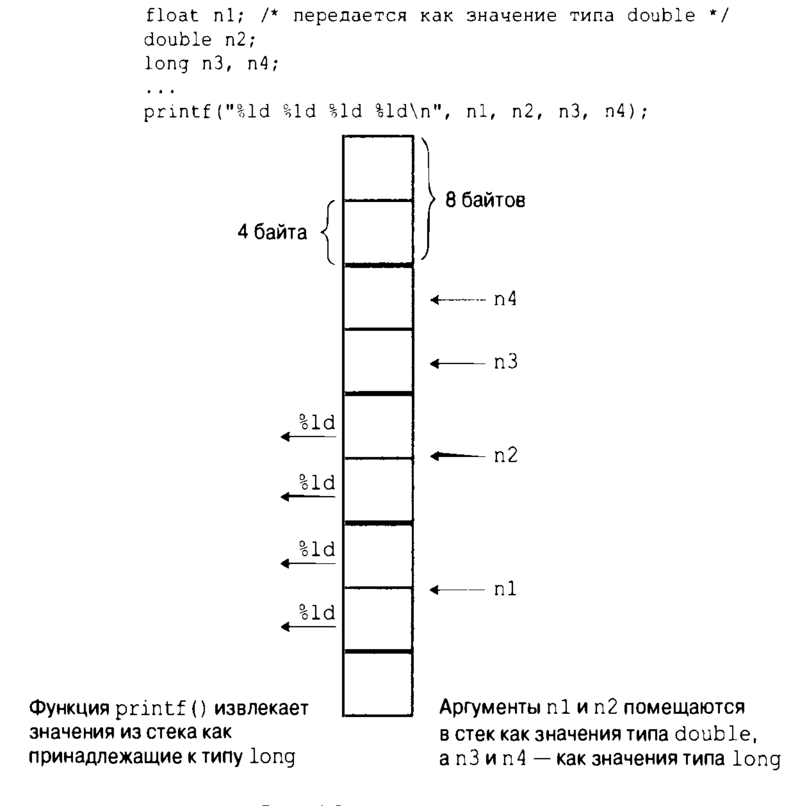


Рис. 4.9. Передача аргументов

Возвращаемое значение функции printf()

Как упоминалось в главе 2, функция в языке C в общем случае имеет возвращаемое значение – это то, что она вычисляет и возвращает в вызывающую программу. Например, в библиотеке C содержится функция sqrt(), которая принимает число в качестве аргумента и возвращает его квадратный корень. Возвращаемое значение может быть присвоено переменной, участвовать в вычислениях, передаваться как аргумент – словом, им можно манипулировать подобно любому другому значению. Функция printf() также имеет возвращаемое значение – количество выведенных символов. Если произошла ошибка вывода, printf() возвратит отрицательное значение. (Некоторые старые версии printf() имели другие возвращаемые значения.)

Возвращаемое значение функции printf() является побочным эффектом ее главной задачи вывода данных и обычно не используется. Единственной причиной работы с возвращаемым значением printf() является необходимость проведения проверки на предмет наличия ошибок вывода. Чаще всего это делается при записи в файл, а не при выводе на экран. Например, если запись на CD- или DVD-диск невозможна из-за его переполнения, программа могла бы предпринимать подходящее действие, такое как выдача звукового сигнала в течение 30 секунд. Тем не менее, прежде чем можно будет реализовать это, необходимо изучить условный оператор if. Простой пример в листинге 4.13 демонстрирует работу с возвращаемым значением.

Листинг 4.13. Программа prntval.c

```
/* prntval.c -- выяснение возвращаемого значения функции printf() */
#include <stdio.h>
int main(void)
{
   int bph2o = 212;
   int rv;

   rv = printf("Вода закипает при %d градусах по Фаренгейту.\n", bph2o);
   printf("Функция printf() вывела %d символов.\n",
          rv);
   return 0;
}
```

Вот вывод этой программы:

```
Вода закипает при 212 градусах по Фаренгейту.
Функция printf() вывела 46 символов.
```

Во-первых, для присваивания возвращаемого значения переменной rv в программе применяется оператор вида rv = printf(...);. Он решает две задачи: выводит информацию и присваивает значение переменной. Во-вторых, обратите внимание, что итоговый результат включает все выведенные символы, в том числе пробелы и невидимый символ новой строки.

Вывод длинных строк

Иногда операторы printf() оказываются слишком длинными, чтобы уместиться в одной строке файла исходного кода. Поскольку в языке C пробельные символы (символы пробела, табуляции и новой строки) игнорируются во всех случаях кроме ситуации, когда они используются для разделения элементов, оператор можно разнести на несколько строк, при условии, что разрывы строк размещаются строго между элементами.

Например, в листинге 4.13 оператор `printf()` находится в двух строках:

```
printf("Функция printf() вывела %d символов.\n",
      rv);
```

В данном случае строка разбита между элементами запятой и `rv`. Чтобы показать читателю кода, что строка имеет продолжение, применяется отступ. Эти избыточные пробелы в C игнорируются.

Однако нельзя разрывать строку внутри кавычек. Предположим, что вы пытаетесь сделать что-то в этом роде:

```
printf("Функция printf() вывела %d
      символов.\n", rv);
```

В таком случае компилятор сообщит об использовании недопустимого символа в строковой константе. Вы можете включить в строку символ `\n`, чтобы обозначить символ новой строки, но не можете иметь внутри строки действительный символ новой строки, сгенерированный нажатием клавиши <Enter> (<Return>).

Когда требуется разбить строку, для этого существуют три возможности, продемонстрированные в листинге 4.14.

Листинг 4.14. Программа `longstrg.c`

```
/* longstrg.c -- вывод длинных строк */
#include <stdio.h>
int main(void)
{
   printf("Вот один из способов вывода ");
   printf("длинной строки.\n");
   printf("Вот второй способ вывода \
длинной строки.\n");
   printf("А вот самый новый способ вывода "
          "длинной строки.\n");  /* ANSI C */
   return 0;
}
```

В результате выполнения программы получается следующий результат:

```
Вот один из способов вывода длинной строки.
Вот второй способ вывода длинной строки.
А вот самый новый способ вывода длинной строки.
```

Первый метод предусматривает применение более одного оператора `printf()`. Поскольку первая выведенная строка не заканчивается символом `\n`, вторая строка продолжается с конца первой.

Второй метод предполагает завершение первой строки комбинацией обратной косой черты и нажатия клавиши <Enter>. Это приводит к тому, что текст на экране начинается с новой строки, но не помещает символ новой строки внутрь строки. В результате строка продолжается на следующей строке экрана. Однако, как видно в листинге, следующая строка экрана должна начинаться с крайней левой позиции. Если вы добавите отступ длиной, скажем, пять пробелов, то эти пять пробелов станут частью строки.

Третий метод, введенный в стандарте ANSI C, называется конкатенацией строк. Если одна строковая константа, заключенная в кавычки, следует за другой такой константой, и они разделены только пробельными символами, то эта комбинация трактуется языком как единая строка. Таким образом, следующие три формы эквивалентны:

```
printf("Привет юным влюбленным, где бы они ни были.");
printf("Привет юным"   "влюбленным" ", где бы они ни были.");
printf("Привет юным влюбленным"
       ", где бы они ни были.");
```

Во всех этих методах вы должны включить в строки любые обязательные пробелы: например, последовательность "юным" "влюбленным" превращается в строку "юнымвлюбленным", а комбинация "юным " "влюбленным" дает в результате "юным влюбленным".

Использование функции scanf()

Теперь давайте перейдем от вывода к вводу и исследуем функцию scanf(). Библиотека С содержит несколько функций ввода, и scanf() является наиболее универсальной из них, т.к. она способна считывать в разных форматах. Разумеется, вводимые с клавиатуры данные являются текстом, поскольку нажатие клавиш приводит к генерации текстовых символов: букв, цифр и знаков препинания. Когда вы хотите ввести, скажем, целое число 2014, вы вводите с клавиатуры символы 2, 0, 1 и 4. Если вы хотите сохранить его как числовое, а не строковое значение, то программа должна выполнить посимвольное преобразование строки в числовое значение — именно это и делает функция scanf(). Она преобразует строковый ввод в разнообразные формы: целые числа, числа с плавающей запятой, символы и строки С. Ее действие противоположно действию функции printf(), которая преобразует целые числа, числа с плавающей запятой, символы и строки С в текст, который затем отображается на экране.

Подобно printf(), в функции scanf() используется управляющая строка, за которой следует список параметров. Управляющая строка указывает целевые типы данных для потока вводимых символов. Главное различие между ними связано со списком аргументов. В функции printf() применяются имена переменных, константы и выражения, а в scanf() — указатели на переменные. К счастью, для использования этой функции знание указателей не требуется. Достаточно запомнить следующие простые правила.

- Если вы используете функцию scanf(), чтобы прочитать значение для переменной одного из базовых типов, предварите имя переменной символом &.
- Если вы применяете функцию scanf() для чтения строки в символьный массив, символ & не нужен.

В листинге 14.15 показана короткая программа, иллюстрирующая эти правила.

Листинг 4.15. Программа input.c

```
// input.c -- ситуации, когда должен использоваться символ &
#include <stdio.h>
int main(void)
{
    int age;            // переменная
    float assets;       // переменная
    char pet[30];       // строка

    printf("Введите информацию о своем возрасте, сумме в банке и любимом животном.\n");
    scanf("%d %f", &age, &assets);  // здесь должен быть указан символ &
    scanf("%s", pet);               // для строкового массива символ & не нужен
    printf("%d $%.2f %s\n", age, assets, pet);

    return 0;
}
```

Ниже приведен пример взаимодействия с программой:

```
Введите информацию о своем возрасте, сумме в банке и любимом животном.
38
92360.88 лама
38 $92360.88 лама
```

При решении, каким образом разделять ввод на отдельные поля, функция scanf() руководствуется пробельными символами (символами новой строки, табуляции и пробела). Она сопоставляет последовательно идущие спецификаторы преобразования с последовательно указанными полями, пропуская промежуточные пробельные символы. Обратите внимание на распространение вводимых данных на две строки. Данные можно было бы вводить как в одной, так и в пяти строках, соблюдая условие, что между отдельными элементами имеется, по меньшей мере, один символ новой строки, пробела или табуляции:

```
Введите информацию о своем возрасте, сумме в банке и любимом животном.
 42

   2121.45

  гуппи
42 $2121.45 гуппи
```

Единственным исключением является спецификатор преобразования %c, который читает каждый следующий символ, даже если он пробельный. Вскоре мы вернемся к этой теме.

В scanf() применяется в основном тот же набор спецификаторов преобразования, что и в printf(). Главное отличие в том, что в функции printf() используются спецификаторы %f, %e, %E, %g и %G для типов float и double, тогда как в scanf() они применяются только для типа float, требуя указания модификатора l для типа double. В табл. 4.6 перечислены основные спецификаторы преобразования, как они описаны в стандарте C99.

Таблица 4.6. Спецификаторы преобразования ANSI C для функции scanf()

Спецификатор преобразования	Значение
%c	Интерпретирует введенные данные как символ
%d	Интерпретирует введенные данные как десятичное целое число со знаком
%e, %f, %g, %a	Интерпретирует введенные данные как число с плавающей запятой (%a появился в C99)
%E, %F, %G, %A	Интерпретирует введенные данные как число с плавающей запятой (%A появился в C99)
%i	Интерпретирует введенные данные как десятичное целое число со знаком
%o	Интерпретирует введенные данные как восьмеричное целое число со знаком
%p	Интерпретирует введенные данные как указатель (адрес)
%s	Интерпретирует введенные данные как строку. Ввод начинается с первого символа, не являющегося пробельным, и включает все символы до следующего пробельного символа
%u	Интерпретирует введенные данные как десятичное целое число без знака
%x, %X	Интерпретирует введенные данные как шестнадцатеричное целое число со знаком

В спецификаторах преобразования можно также использовать модификаторы, перечисленные в табл. 4.6. Модификаторы размещаются между знаком % и буквой преобразования. В случае указания в спецификаторе нескольких модификаторов они должны появляться в том же самом порядке, как они описаны в табл. 4.7.

Таблица 4.7. Модификаторы преобразования функции scanf()

Модификатор	Значение
*	Подавляет присваивание (пояснения приведены далее в главе). Пример: "%*d"
цифра (*цифры*)	Максимальная ширина поля. Ввод прекращается, когда достигнута максимальная ширина поля или если обнаружен первый пробельный символ (в зависимости от того, что произойдет раньше). Пример: "%10s"
hh	Читает целое число как signed char или unsigned char. Примеры: "%hhd", "%hhu"
ll	Читает целочисленное число как long long или как long long без знака (стандарт С99). Примеры: "%lld", "%llu"
h, l или L	Спецификаторы "%hd" и "%hi" указывают, что значение будет сохранено с типом short int. Спецификаторы "%ho", "%hx" и "%hu" определяют, что значение будет сохранено с типом unsigned short int. Спецификаторы "%ld" и "%li" указывают, что значение будет сохранено с типом long. Спецификаторы "%lo", "%lx" и "%lu" определяют, что значение будет сохранено с типом unsigned long. Спецификаторы "%le", "%lf" и "%lg" указывают, что значение будет сохранено с типом double. Использование модификатора L вместо l в сочетании с e, f и g определяет, что значение будет сохранено с типом long double. В отсутствие этих модификаторов d, i, o и x указывают на тип int, а e, f и g — на тип float
j	Когда за ним следует спецификатор целочисленного преобразования, указывает на использование типа intmax_t или uintmax_t (стандарт С99). Примеры: "%jd" "%ju"
z	Когда за ним следует спецификатор целочисленного преобразования, указывает на использование типа, возвращенного операцией sizeof (стандарт С99). Примеры: "%zd" "%zo"
t	Когда за ним следует спецификатор целочисленного преобразования, указывает на использование типа, служащего для представления разницы между двумя указателями (стандарт С99). Примеры: "%td", "%tx".

Как видите, применение спецификаторов преобразования может быть весьма разнообразным, причем в таблицах были показаны далеко не все средства. Эти средства связаны главным образом с облегчением чтения выбранных данных из жестко форматированных источников, таких как перфокарты или другие записи данных. Поскольку в книге функция scanf() используется в основном как удобный инструмент для интерактивной передачи данных программе, такие экзотические особенности здесь не обсуждаются.

Обработка ввода функцией `scanf()`

Давайте более подробно рассмотрим, как функция `scanf()` считывает поток вводимых данных. Предположим, что вы применяете спецификатор `%d`, чтобы прочитать целое число. Функция `scanf()` начинает читать поток ввода по одному символу за раз. Она пропускает пробельные символы (символы пробела, табуляции и новой строки) до тех пор, пока не натолкнется на символ, отличный от пробельного. Поскольку функция `scanf()` пытается прочитать целое число, она ожидает обнаружить цифровой символ или, возможно, знак (+ или –). Встретив цифру или знак, она запоминает этот символ и считывает следующий. Если это цифра, она сохраняет ее и читает следующий символ. Функция `scanf()` продолжает чтение и сохранение символов, пока не столкнется с нецифровым символом. Тогда функция приходит к заключению, что она достигла конца очередного целого числа. Функция `scanf()` помещает этот нецифровой символ обратно в поток ввода. Это означает, что в следующий раз, когда программа приступит к чтению потока ввода, она начнет его с ранее отклоненного нецифрового символа. Наконец, функция `scanf()` вычисляет числовое значение, соответствующее считанным ею цифрам (и, возможно, знаку), и заносит это значение в указанную переменную.

Если вы используете ширину поля, функция `scanf()` прекращает чтение при достижении конца поля или на первом пробельном символе, в зависимости от того, что произойдет раньше.

А что случится, если первый отличный от пробельного символ представляет собой, скажем, символ `A`, а не цифру? В таком случае функция `scanf()` тут же останавливается и помещает символ `A` (или другой) обратно в поток ввода. Указанной переменной значение не присваивается, и в следующий раз, когда программа будет читать поток ввода, она снова начнет его с `A`. Если вы применяете в программе только спецификаторы `%d`, то функция `scanf()` никогда не продвинется дальше этого символа `A`. Кроме того, если вы используете `scanf()` с несколькими спецификаторами, то язык С требует, чтобы функция прекращала чтение потока ввода при первом же отказе.

Чтение потока ввода с применением других числовых спецификаторов происходит так же, как в случае спецификатора `%d`. Главное различие между ними заключается в том, что функция `scanf()` может распознавать больше символов в качестве части числа. Например, спецификатор `%x` требует, чтобы функция `scanf()` распознавала символы a–f и A–F как шестнадцатеричные цифры. Спецификаторы с плавающей запятой требуют, чтобы функция `scanf()` распознавала десятичные точки, экспоненциальную форму записи и новую р-нотацию.

Если вы используете спецификатор `%s`, то допускается любой символ, отличный от пробельного, поэтому функция `scanf()` пропускает пробельные символы до появления первого непробельного символа, после чего сохраняет все непробельные символы вплоть до следующего появления пробельного символа. Это означает, что спецификатор `%s` заставляет функцию `scanf()` читать одиночное слово, т.е. строку, которая не содержит пробельных символов. В случае указания ширины поля `scanf()` прекращает чтение при достижении конца поля или на первом пробельном символе, в зависимости от того, что произойдет раньше. С помощью ширины поля нельзя заставить функцию `scanf()` читать более одного слова для одного спецификатора `%s`. И последний момент: когда функция `scanf()` помещает строку в назначенный массив, она добавляет завершающий символ `'\0'` с тем, чтобы сделать содержимое массива строкой С.

Если вы задаете спецификатор `%c`, то все вводимые символы запоминаются в исходном виде. Если следующим вводимым символом является символ пробела или новой строки, то он и присваивается указанной переменной; пробельные символы не пропускаются.

В действительности функция scanf() не относится к числу наиболее часто используемых функций ввода в С. Она рассматривается здесь по причине своей универсальности (т.к. умеет читать все многообразие типов данных). В языке С доступно несколько других функций ввода вроде getchar() и fgets(), которые лучше подходят для решения специфичных задач, например, чтения одиночных символов или чтения строк, содержащих пробелы. Некоторые из этих функций будут рассмотрены в главах 7, 11 и 13. А пока для ввода целого числа или десятичной дроби, символа или строки можете применять функцию scanf().

Обычные символы в строке формата

Функция scanf() позволяет помещать в строку формата обычные символы. Вводимая строка должна обеспечивать точное совпадение для обычных символов, отличных от пробельных. Например, предположим, что вы непредумышленно поместили запятую между двумя спецификаторами:

```
scanf("%d,%d", &n, &m);
```

Функция scanf() интерпретирует эту строку так, что вам придется набрать число, затем запятую и, наконец, второе число. То есть вы должны вводить два целых числа следующим образом:

```
88,121
```

Поскольку в строке формата запятая находится непосредственно после спецификатора %d, ее требуется набирать сразу после числа 88. Тем не менее, с учетом того, что scanf() пропускает пробельные символы, предшествующие целому числу, при вводе можно было бы набрать пробел или символ новой строки. Другими словами, показанные ниже варианты также будут приемлемыми:

```
88, 121
```

и

```
88,
121
```

Пробел в строке формата означает необходимость пропуска любых пробельных символов перед следующим элементом ввода. Например, оператор

```
scanf("%d,%d", &n, &m);
```

принял бы любую из следующих входных строк:

```
88,121
88 ,121
88 ,  121
```

Обратите внимание, что концепция “любые пробельные символы” охватывает также специальный случай отсутствия пробельных символов.

За исключением %c все остальные спецификаторы автоматически пропускают пробельный символ, предваряющий вводимое значение, так что оператор scanf("%d%d", &n, &m) ведет себя точно так же, как scanf("%d %d", &n, &m). Для спецификатора %c наличие или отсутствие символа пробела в строке формата не вносит никакой разницы. Например, если в строке формата спецификатору %c предшествует пробел, то функция scanf() пропускает все до появления первого непробельного символа. Таким образом, оператор scanf("%c", &ch) читает первый значащий символ, с которым сталкивается во введенных данных, а scanf(" %c", &ch) читает первый встреченный непробельный символ.

Возвращаемое значение функции scanf()

Функция scanf() возвращает количество элементов, которые она успешно прочитала. Если не прочитано ни одного элемента, как бывает в случае набора нечисловой строки, в то время когда scanf() ожидает число, возвращается 0. При обнаружении условия, называемого "конец файла" ("end of file"), функция возвращает EOF. (EOF — это специальное значение, определенное в файле stdio.h. Обычно с помощью директивы #define константе EOF присваивается значение -1.) Мы рассмотрим признак конца файла в главе 6, а вопросы использования возвращаемого значения функции scanf() — позже в этой главе. После изучения операторов if и while вы сможете задействовать возвращаемое значение scanf() для обнаружения и обработки несогласованного ввода.

Модификатор * в функциях printf() и scanf()

И в printf(), и в scanf() модификатор * можно применять для изменения значения спецификатора, но делается это по-разному. Для начала давайте рассмотрим использование модификатора * в функции printf().

Предположим, что вы не хотите фиксировать ширину поля заранее, но желаете, чтобы ее определила сама программа. Это можно сделать, указав вместо числа, задающего ширину поля, модификатор *, но понадобится также добавить аргумент для сообщения функции, какой должна быть ширина поля. То есть при наличии спецификатора преобразования %*d список аргументов должен содержать значение для модификатора * *и* значение для d. Такой метод можно применять также со значениями с плавающей запятой, чтобы указывать точность и ширину поля. В листинге 4.16 приведен небольшой пример, демонстрирующий, как все это работает.

Листинг 4.16. Программа varwid.c

```
/* varwid.c -- использование поля вывода переменной ширины */
#include <stdio.h>
int main(void)
{
   unsigned width, precision;
   int number = 256;
   double weight = 242.5;

   printf("Введите ширину поля:\n");
   scanf("%d", &width);
   printf("Значение равно: %*d:\n", width, number);
   printf("Теперь введите ширину и точность:\n");
   scanf("%d %d", &width, &precision);
   printf("Вес = %*.*f\n", width, precision, weight);
   printf("Готово!\n");

   return 0;
}
```

Переменная width определяет ширину поля, а переменная number — это число, которое должно быть выведено. Поскольку модификатор * предшествует d в спецификаторе, значение width находится раньше значения number в списке параметров функции printf(). Подобным образом значения width и precision предоставляют необходимую информацию для форматирования для вывода значения weight. Взгляните на пример выполнения этой программы:

```
Введите ширину поля:
6
Значение равно:   256:
Теперь введите ширину и точность:
8 3
Вес =  242.500
Готово!
```

В этом случае ответом на первый вопрос было число 6, поэтому 6 используется для ширины поля. Второй ответ привел к установке ширины поля в 8 и отображению 3 цифр справа от десятичной точки. В целом программа могла бы принять решение относительно значений для этих переменных после анализа значения weight.

В случае функции scanf() модификатор * служит совершенно другой цели. Когда он помещен между символом % и буквой спецификатора, модификатор * вынуждает функцию пропускать соответствующий ввод. В листинге 4.17 предоставлен пример.

Листинг 4.17. Программа skip2.c

```
/* skiptwo.c -- пропускает первые два целых числа в потоке ввода */
#include <stdio.h>
int main(void)
{
   int n;
   printf("Введите три целых числа:\n");
   scanf("%*d %*d %d", &n);
   printf("Последним целым числом было %d\n", n);
   return 0;
}
```

Оператор scanf() в листинге 4.17 указывает программе на необходимость пропуска двух целых чисел и копирования третьего целого числа в переменную n. Ниже показан пример выполнения этой программы:

```
Введите три целых числа:
2013 2014 2015
Последним целым числом было 2015
```

Такая возможность пропуска полезна, если программе, например, требуется читать конкретный столбец в файле, в котором данные организованы в виде унифицированных столбцов.

Советы по использованию функции printf()

Указание фиксированной ширины полей полезно, когда вы хотите выводить столбцы данных. Поскольку стандартной шириной поля является просто ширина числа, многократное применение оператора следующего вида:

```
printf("%d %d %d\n", val1, val2, val3);
```

приводит к генерации неровных столбцов, если числа в столбце имеют разные размеры. Например, вывод может выглядеть так:

```
12 234 1222
4 5 23
22334 2322 10001
```

(Здесь предполагается, что значения переменных изменялись между выполнением операторов printf().)

Выводу можно придать более аккуратный вид, если использовать достаточно большую фиксированную ширину поля.

Например, применение оператора

```
printf("%9d %9d %9d\n", val1, val2, val3);
```

дает следующий вывод:

```
   12      234     1222
    4        5       23
22334     2322    10001
```

Помещение пробела между одним спецификатором преобразования и следующим за ним спецификатором предотвращает перекрытие одного числа следующим, даже если оно не умещается в собственное поле. Так происходит потому, что обычные символы, указанные в управляющей строке, выводятся всегда, и пробелы тому не исключение.

С другой стороны, если число должно быть внедрено внутрь фразы, часто удобно определить поле таким же или меньшим по размеру, чем ожидаемая ширина числа. Это обеспечит размещение числа без нежелательных пробелов. Например, оператор

```
printf("Каунт Беппо пробежал %.2f миль за 3 часа.\n", distance);
```

выводит следующую фразу:

```
Каунт Беппо пробежал 10.22 миль за 3 часа.
```

Изменение спецификатора преобразования на `%10.2f` привело бы к такому результату:

```
Каунт Беппо пробежал      10.22 миль за 3 часа.
```

Выбор локали

В США и многих других странах мира для отделения целочисленной части от дробной используется точка, как в 3.14159. В то же время во множестве других стран для этого применяется запятая, как в 3,14159. Вы могли заметить, что спецификаторы функций `printf()` и `scanf()` не предусматривают формат с использованием запятой. Однако в языке С не забыли о других странах. Как описано в разделе V приложения Б, язык С поддерживает понятие *локали*. Это предоставляет программе С возможность выбора конкретной локали. Например, можно было бы указать локаль Нидерландов, тогда функции `printf()` и `scanf()` использовали бы локальное соглашение (в данном случае запятую) при отображении и чтении значений с плавающей запятой. Кроме того, после указания данной среды соглашение в отношении запятой применялось бы для чисел, появляющихся в коде:

```
double pi = 3,14159;  // локаль Нидерландов
```

Стандарт С требует использования одной из двух локалей: `"C"` и `""`. По умолчанию программы применяют локаль `"C"`, что по существу соответствует принятому в США способу представления чисел. Локаль `""` подразумевает локаль, используемую в конкретной системе. В принципе, она может совпадать с локалью `"C"`. На практике операционные системы, такие как Unix, Linux и Windows, предоставляют обширные списки вариантов локалей, однако эти списки могут различаться.

Ключевые понятия

В языке С тип `char` представляет одиночный символ. Для представления последовательности символов в С применяется символьная строка. Одной из форм строки является символьная константа, в которой символы заключены в двойные кавычки, например, `"Удачи, друзья!"`. Вы можете хранить строку в массиве символов, который состоит из смежных байтов памяти. Символьные строки, выраженные в виде символьной константы или сохраненные в символьном массиве, завершаются скрытым символом, который называется *нулевым* символом.

Числовые константы целесообразно представлять в программе символически, либо посредством директивы #define, либо с помощью ключевого слова const. Символические константы делают программу более читабельной и легкой для сопровождения и модификации.

Стандартные функции ввода и вывода scanf() и printf() языка С используют систему, при которой вы должны сопоставлять спецификаторам внутри первого аргумента значения в последующих аргументах. Сопоставление, скажем, спецификатора int, такого как %d, со значением float приведет к непредсказуемым результатам. Необходимо внимательно следить за тем, чтобы количество и типы спецификаторов соответствовали остальным аргументам функции. Для scanf() не забывайте предварять имена переменных операцией взятия адреса (&).

Пробельные символы (символы табуляции, пробела и новой строки) играют критически важную роль в том, как scanf() просматривает вводимые данные. За исключением режима, устанавливаемого спецификатором %c (который читает только следующий символ), при чтении входных данных функция scanf() пропускает пробельные символы вплоть до первого непробельного символа. Затем она продолжает чтение символов до тех пор, пока не встретит пробельный символ либо символ, не подходящий для типа, для которого считывается значение. Давайте посмотрим, что происходит при вводе одной и той же информации в разных режимах ввода функции scanf(). Начнем со следующей входной строки:

```
-13.45e12#  0
```

Сначала предположим, что применяется режим %d; функция scanf() прочитает три символа (-13) и остановится на точке как на следующем входном символе. Затем scanf() преобразует последовательность символов -13 в соответствующее целочисленное значение и сохранит его в целевой переменной типа int. В режиме %f функция scanf() прочитает символы -13.45E12 и остановится на символе #, оставив его для последующего ввода. Далее она преобразует последовательность символов -13.45E12 в соответствующее значение с плавающей запятой и сохранит его в целевой переменной типа float. В случае режима %s функция scanf() прочитает последовательность символов -13.45E12# и остановится на пробеле как на следующем символе для ввода. Затем она сохранит коды всех этих десяти символов в целевом символьном массиве, добавив в конец нулевой символ. При чтении этой же строки в режиме %c функция scanf() прочитает и сохранит первый символ, т.е. пробел.

Резюме

Строка — это последовательность символов, трактуемая как отдельная единица. В языке С строка представлена последовательностью символов, завершающейся нулевым символом, ASCII-код которого равен 0. Строки могут храниться в символьных массивах. Массив — это последовательность элементов, имеющих один и тот же тип. Чтобы объявить массив name, содержащий 30 элементов типа char, используйте следующий оператор:

```
char name[30];
```

Позаботьтесь о выделении такого количества элементов, которого достаточно для хранения всей строки, включая нулевой символ.

Строковые константы создаются путем заключения строки в двойные кавычки:

```
"Это пример строковой константы"
```

Функцию strlen() (объявленную в заголовочном файле string.h) можно применять для выяснения длины строки (без учета завершающего нулевого символа). Функция scanf(), будучи вызванной вместе со спецификатором %s, может использоваться для чтения строк, состоящих из одного слова.

Препроцессор языка С ищет в исходном тексте программы директивы препроцессора, которые начинаются с символа #, и действует согласно им до начала процесса компиляции программы. Директива #include заставляет препроцессор добавить содержимое другого файла в текущий файл там, где эта директива находится. Директива #define позволяет определять символические константы. В заголовочных файлах limits.h и float.h директива #define применяется для определения набора констант, представляющих разнообразные свойства целочисленных типов и типов с плавающей запятой. Для создания символических констант можно также использовать модификатор const.

Функции printf() и scanf() предоставляют универсальную поддержку для ввода и вывода. В каждой из них применяется управляющая строка, содержащая вложенные спецификаторы преобразования, которые указывают количество и типы элементов данных, подлежащих чтению или выводу. Вдобавок можно использовать спецификаторы преобразования для управления внешним видом вывода: шириной поля, количеством десятичных позиций и выравниванием в рамках поля.

Вопросы для самоконтроля

Ответы на эти вопросы находятся в приложении А.

1. Запустите программу из листинга 4.1 еще раз, и когда программа запросит ввод имени, введите имя и фамилию. Что происходит? Почему?

2. Предположим, что каждый из следующих примеров является частью завершенной программы. Что будет выводить каждая такая часть?

   а. `printf("Он продал эту картину за $%2.2f.\n", 2.345e2);`

   б. `printf("%c%c%c\n", 'H', 105, '\41');`

   в.
   ```
   #define Q "Его Гамлет был хорош, и без намека на вульгарность."
   printf("%s\nсодержит %d символов.\n", Q, strlen(Q));
   ```

   г. `printf("Является ли %2.2e тем же, что и %2.2f?\n", 1201.0, 1201.0);`

3. Какие изменения необходимо сделать в пункте в) второго вопроса, чтобы строка Q была выведена в двойных кавычках?

4. Попробуйте найти ошибку в следующем коде:

```
define B booboo
define X 10
main(int)
{
    int age;
    char name;
    printf{"Введите свое имя.");
    scanf("%s", name);
    printf("Хорошо, %C, а сколько вам лет?\n", name);
    scanf("%f", age);
    xp = age + X;
    printf ("Неужели, %s! Вам должно быть, по меньшей мере, %d.\n", B, xp);
    rerun 0;
}
```

5. Предположим, что программа начинается так:

```
#define BOOK "Война и мир"
int main(void)
{
   float cost = 12.99;
   float percent = 80.0;
```

Напишите оператор `printf()`, который использует `BOOK`, `cost` и `percent` для следующего вывода:

```
Данный экземпляр книги "Война и мир" стоит $12.99.
Это 80% от цены в прайс-листе.
```

6. Какие спецификаторы преобразования вы бы использовали, чтобы вывести следующие данные?
   а. Десятичное целое число с шириной поля, равной количеству цифр этого числа.
   б. Шестнадцатеричное целое число в форме 8A с шириной поля 4 символа.
   в. Число с плавающей запятой в форме 232.346 с шириной поля 10 символов.
   г. Число с плавающей запятой в форме 2.33e+002 с шириной поля 12 символов.
   д. Строку, выровненную по левому краю внутри поля шириной 30 символов.
7. Какие спецификаторы преобразования вы бы использовали, чтобы вывести следующие данные?
   а. Целое число типа `unsigned long` в поле шириной 15 символов.
   б. Шестнадцатеричное целое число в форме 0x8a в поле шириной 4 символа.
   в. Число с плавающей запятой в форме 2.33E+02 в поле шириной 12 символов с выравниванием по левому краю поля.
   г. Число с плавающей запятой в форме +232.346 в поле шириной 10 символов.
   д. Первые 8 символов строки в поле шириной 8 символов.
8. Какие спецификаторы преобразования вы бы использовали, чтобы вывести следующие данные?
   а. Десятичное целое число, имеющее минимум 4 цифры, в поле шириной 6 символов.
   б. Восьмеричное целое число в поле, ширина которого будет указываться в списке аргументов.
   в. Символ в поле шириной 2 символа.
   г. Число с плавающей запятой в форме +3.13 в поле с шириной, которая равна количеству символов в этом числе.
   д. Первые пять символов в строке, выровненной по левому краю поля шириной 7 символов.
9. Для каждой из следующих входных строк напишите оператор `scanf()`, чтобы прочитать их. Объявите также переменные или массивы, используемые в операторе.
   а. `101`
   б. `22.32 8.34E-09`
   в. `linguini`
   г. `catch 22`
   д. `catch 22` (но пропустить `catch`)

10. Что такое пробельный символ?

11. В чем заключается ошибка в следующем операторе, и как ее можно исправить?

```
printf("Тип double состоит из %z байтов..\n", sizeof (double));
```

12. Предположим, что в своих программах вы хотели бы использовать круглые скобки вместо фигурных. Насколько хорошо бы работали следующие конструкции?

```
#define ( {
#define ) }
```

Упражнения по программированию

1. Напишите программу, которая запрашивает имя и фамилию, а затем выводит их в формате *фамилия, имя.*

2. Напишите программу, которая запрашивает имя и выполняет с ним следующие действия.

   а. Выводит его заключенным в двойные кавычки.

   б. Выводит его в поле шириной 20 символов, при этом все поле заключается в кавычки, а имя выравнивается по правому краю поля.

   в. Выводит его с левого края поля шириной 20 символов, при этом все поле заключается в кавычки.

   г. Выводит его в поле шириной, на три символа превышающем длину имени.

3. Напишите программу, которая читает число с плавающей запятой и выводит его сначала в десятичной, а затем в экспоненциальной форме. Предусмотрите вывод в следующих форматах (количество цифр показателя степени в вашей системе может быть другим).

   а. Вводом является `21.3` или `2.1e+001`.

   б. Вводом является `+21.290` или `2.129E+001`.

4. Напишите программу, которая запрашивает рост в дюймах и имя, после чего отображает полученную информацию в следующей форме:

   ```
   Ларри, ваш рост составляет 6.208 футов
   ```

   Используйте тип `float`, а также операцию деления `/`. Если хотите, можете запрашивать рост в сантиметрах и отображать его в метрах.

5. Напишите программу, которая запрашивает скорость загрузки в мегабитах в секунду и размер файла в мегабайтах. Программа должна вычислять время загрузки файла. Имейте в виду, что в данном случае один байт равен восьми битам. Используйте тип `float`, а также операцию деления `/`. Программа должна выводить все три значения (скорость загрузки, размер файла и время загрузки) с отображением двух цифр справа от десятичной точки, как в следующем выводе:

   ```
   При скорости загрузки 18.12 мегабит в секунду файл размером 2.20 мегабайт
   загружается за 0.97 секунд(ы).
   ```

6. Напишите программу, которая запрашивает имя пользователя и его фамилию. Сделайте так, чтобы она выводила введенные имена в одной строке и количество символов в каждом слове в следующей строке. Выровняйте каждое количество символов по окончанию соответствующего имени, как показано ниже:

   ```
   Иван Петров
      4      6
   ```

Затем сделайте так, чтобы программа выводила ту же самую информацию, но с количеством символов, выровненным по началу каждого слова:

```
Иван Петров
4    6
```

7. Напишите программу, которая присваивает переменной типа double значение 1.0/3.0 и переменной типа float значение 1.0/3.0. Отобразите каждый результат три раза: в первом случае с четырьмя цифрами справа от десятичной точки, во втором случае с двенадцатью цифрами и в третьем случае с шестнадцатью цифрами. Включите также в программу заголовочный файл float.h и выведите значения FLT_DIG и DBL_DIG. Согласуются ли выведенные значения со значением 1.0/0.3?

8. Напишите программу, которая предлагает пользователю ввести количество преодоленных миль и количество галлонов израсходованного бензина. Затем эта программа должна рассчитать и отобразить на экране количество миль, пройденных на одном галлоне горючего, с одним знаком после десятичной точки. Далее, учитывая, что один галлон равен приблизительно 3.785 литра, а одна миля составляет 1.609 километра, программа должна перевести значение в милях на галлон в литры на 100 километров (обычную европейскую меру измерения потребления горючего) и вывести результат с одним знаком после десятичной точки. Обратите внимание, что в США принято измерять пробег на единицу горючего (чем выше, тем лучше), в то время как в Европе принято измерять расход топлива на единицу расстояния (чем ниже, тем лучше). Применяйте для этих двух коэффициентов преобразования символические константы (определенные с помощью const или #define).

5

Операции, выражения и операторы

В ЭТОЙ ГЛАВЕ...

- Ключевые слова: while, typedef
- Операции: = - * / % ++ — (тип)
- Разнообразные операции языка С, включая используемые для распространенных арифметических действий
- Приоритеты операций и значение терминов оператор и выражение
- Удобный цикл while
- Составные операторы, автоматическое преобразование типов и приведение типов
- Написание функций, принимающих аргументы

Теперь, когда вы ознакомились со способами представления данных, давайте приступим к исследованию методов обработки данных. Для этих целей в языке C предлагается множество разнообразных операций. Вы можете выполнять арифметические действия, сравнивать значения, обновлять значения переменных, логически объединять отношения и делать многое другое. Начнем с базовых арифметических действий — сложения, вычитания, умножения и деления.

Другим аспектом обработки данных является такая организация программ, которая обеспечит выполнение ими правильных действий в должном порядке. Язык C обладает несколькими языковыми средствами, которые помогают решить эту задачу. Одним из таких средств является цикл, и в этой главе вы получите о нем первое представление. Цикл позволяет повторять действия и делать программу более интересной и мощной.

Введение в циклы

В листинге 5.1 показана демонстрационная программа, выполняющая несложные арифметические действия для вычисления длины ступни в дюймах, для которой подходит мужская обувь размера 9 (применяемого в США). Чтобы вы лучше смогли оценить преимущества циклов, в этой первой версии программы иллюстрируются ограничения программирования без использования циклов.

Листинг 5.1. Программа shoes1.c

```
/* shoes1.c -- преобразует размер обуви в дюймы */
#include <stdio.h>
#define ADJUST 7.31              // один из видов символической константы
int main(void)
{
   const double SCALE = 0.333;  // еще один вид символической константы
   double shoe, foot;

   shoe = 9.0;
   foot = SCALE * shoe + ADJUST;
   printf("Размер обуви (мужской)    длина ступни\n");
   printf("%10.1f %20.2f дюймов\n", shoe, foot);

   return 0;
}
```

Ниже приведен вывод:

```
Размер обуви (мужской)    длина ступни
       9.0               10.31 дюймов
```

Программа демонстрирует два способа создания символических констант и в ней применяются умножение и сложение. Она принимает размер обуви (если вы носите размер 9) и сообщает длину вашей ступни в дюймах. Вы можете заявить, что в состоянии решить эту задачу вручную (или на калькуляторе) быстрее, чем будет произведен ввод этого кода с клавиатуры. Это правильное замечание. Написание одноразовой программы, обрабатывающей единственный размер обуви, является напрасной тратой времени и сил. Программу можно сделать более полезной, реализовав ее как интерактивную, но и в этом случае потенциал компьютера не будет задействован в полной мере.

Необходим какой-то способ, который позволил бы заставить компьютер выполнять повторяющиеся вычисления для заданной последовательности размеров обуви. В конце концов, это одна из причин применения компьютеров в арифметических вычислениях. Язык С предлагает несколько методов реализации повторяющихся вычислений, и здесь мы рассмотрим один из них. Этот метод, называемый *циклом* `while`, позволит более эффективно использовать операции. В листинге 5.2 приведен усовершенствованный вариант программы для определения длины стопы по размеру обуви.

Листинг 5.2. Программа shoes2.c

```
/* shoes2.c -- вычисляет длину стопы для нескольких размеров обуви */
#include <stdio.h>
#define ADJUST 7.31              // один из видов символической константы
int main(void)
{
   const double SCALE = 0.333;  // еще один вид символической константы
   double shoe, foot;
   printf("Размер обуви (мужской)   длина ступни\n");
   shoe = 3.0;
   while (shoe < 18.5)     /* начало цикла while */
   {                       /* начало блока */
      foot = SCALE * shoe + ADJUST;
      printf("%10.1f %20.2f дюймов\n", shoe, foot);
      shoe = shoe + 1.0;
   }                       /* конец блока  */
   printf("Если обувь подходит, носите ее.\n");
   return 0;
}
```

Вот как выглядит сжатая версия вывода программы `shoes2.c`:

```
Размер обуви (мужской)   длина ступни\n
       3.0                  8.31 дюймов
       4.0                  8.64 дюймов
       5.0                  8.97 дюймов
       6.0                  9.31 дюймов

      16.0                 12.64 дюймов
      17.0                 12.97 дюймов
      18.0                 13.30 дюймов
Если обувь подходит, носите ее.
```

(Те, кто всерьез интересуется размерами обуви, должны иметь в виду, что в этой программе сделано нереалистичное предположение о наличии рациональной и унифицированной системы размеров обуви. В реальности системы размеров могут отличаться.)

Давайте посмотрим, как работает цикл `while`. Когда управление впервые доходит до оператора `while`, выполняется проверка того, принимает ли условие в круглых скобках значение `true`. В этом случае выражение условия имеет следующий вид:

```
shoe < 18.5
```

Символ < означает "меньше чем". Переменная `shoe` инициализирована значением `3.0`, что определенно меньше чем `18.5`. Поэтому условие равно `true` и управление переходит на следующий оператор, который преобразует размер в дюймы. Затем программа выводит результат. Следующий оператор увеличивает значение `shoe` на `1.0`, делая его равным `4.0`:

```
shoe = shoe + 1.0;
```

В этой точке управление возвращается к порции while, чтобы проверить условие. Но почему именно в этой точке? Причина в том, что в следующей строке находится закрывающая фигурная скобка (}), а код использует пару таких скобок ({}) для обозначения границ цикла while. Операторы, находящиеся между двумя фигурными скобками, повторяются. Раздел программы внутри фигурных скобок и сами фигурные скобки называются *блоком*. А теперь вернемся к программе. Значение 4.0 меньше 18.5, поэтому все операторы, заключенные в фигурные скобки (блок), следующие за while, повторяются. (На компьютерном жаргоне можно сказать, что программа "проходит в цикле" по этим операторам.) Это продолжается до тех пор, пока переменная shoe не достигнет значения 19.0. Тогда условие

```
shoe < 18.5
```

получает значение false, потому что 19.0 не меньше 18.5. Как только это произойдет, управление передается первому оператору, следующему за циклом while. В данном случае им является финальный оператор printf().

Рассмотренную программу можно легко модифицировать для выполнения других преобразований. Например, установив SCALE в 1.8 и ADJUST в 32.0, вы получите программу, которая преобразует значение температуры по Цельсию в значение по Фаренгейту. Присвоив SCALE значение 0.6214 и ADJUST – 0, вы реализуете преобразование километров в мили. Естественно, понадобится также соответствующим образом изменить выводимые сообщения. Цикл while предоставляет в ваше распоряжение удобное и гибкое средство управления внутри программы. Теперь давайте перейдем к ознакомлению с фундаментальными операциями, которые вы можете применять в своих программах.

Фундаментальные операции

Для представления арифметических действий в языке С используются *операции*. Например, операция + вызывает сложение двух значений, находящиеся по обе стороны символа операции. Если термин *операция* кажется вам странным, подумайте о том, что вещи такого рода должны как-то называться. "Операция" представляется более удачным вариантом, чем, скажем, "эта вещь" или "арифметический транзактор". Теперь рассмотрим операции, применяемые для базовой арифметики: =, +, -, * и /. (В языке С операция возведения в степень отсутствует. Тем не менее, библиотека стандартных математических функций С предлагает для этих целей функцию pow(). Например, pow(3.5, 2.2) возвращает значение 3.5, возведенное в степень 2.2.)

Операция присваивания: =

В языке С знак = не означает "равно". Вместо этого им обозначается операция присваивания значения. Например, следующий оператор присваивает значение 2002 переменной по имени bmw:

```
bmw = 2002;
```

То есть элемент, расположенный слева от знака =, представляет собой имя переменной, а элемент справа – значение, присваиваемое этой переменной. Символ = называется *операцией присваивания*. Еще раз: ни в коем случае не думайте, что эта строка гласит: "переменная bmw равна 2002". Взамен читайте ее так: "присвоить переменной bmw значение 2002". Для этой операции действие происходит справа налево.

Возможно, такое различие между именем переменной и ее значением выглядит запутанным, но взгляните на следующий типичный пример оператора:

```
i = i + 1;
```

С точки зрения математики этот оператор не имеет смысла. После прибавления 1 к конечному числу результат не может быть "равен" исходному числу, однако как компьютерный оператор присваивания он совершенно корректен. Его смысл таков: "извлечь значение переменной по имени i, добавить 1 к этому значению и затем присвоить новое значение переменной i" (рис. 5.1).

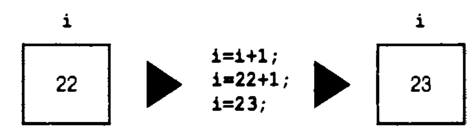


Рис. 5.1. Оператор `i = i + 1;`

Оператор наподобие

```
2002 = bmw;
```

в C не имеет смысла (и, разумеется, не является допустимым), т.к. 2002 — это то, что в языке C называется *r-значением*, которое в данном случае представляет собой просто литеральную константу. Присвоить значение константе невозможно; она уже *является* эквивалентом значения. Следовательно, не забывайте, что элемент слева от знака = должен быть именем переменной. В действительности левая сторона должна ссылаться на ячейку памяти. Простейший способ предполагает использование имени переменной, но как вы увидите позже, для этого можно применять "указатель". В общем случае для пометки сущностей, которым можно присваивать значения, в C используется термин *модифицируемое l-значение*. Возможно, данный термин не особенно понятен, поэтому давайте введем некоторые определения.

Немного терминологии: объекты данных, l-значения, r-значения и операнды

Рассмотрим оператор присваивания. Его назначение заключается в том, чтобы сохранить значение в ячейке памяти. *Объект данных* — это общий термин для обозначения области хранения данных, которая может применяться для удержания значений. Для обозначения этой концепции в стандарте языка C используется термин *объект*. Один из способов идентификации объекта предполагает применение имени переменной. Но, как со временем вы узнаете, для идентификации объекта существуют и другие способы. Например, вы могли бы указать элемент массива, член структуры или воспользоваться выражением указателя, которое включает в себя адрес объекта. Термин *l-значение* в C применяется для обозначения любого имени или выражения подобного рода, идентифицирующего конкретный объект данных. Объект относится к фактической области хранения данных, но l-значение — это метка, которая используется для определения местоположения этой области памяти.

На заре развития языка C именование чего-либо l-значением происходило при двух обстоятельствах.

1. Оно указывало объект, тем самым ссылаясь на адрес в памяти.
2. Оно могло применяться слева от символа операции присваивания; отсюда и буква "l" (от "left" — "левая часть") в названии *l-значение*.

Но затем в языке C появился модификатор const. Он позволяет создавать объект, но такой, значение которого не может изменяться. В итоге идентификатор const соответствует первому из двух описанных выше обстоятельств, но не второму. В настоящее время стандарт продолжает использовать понятие l-значения для любого

выражения, идентифицирующего объект, хотя некоторые l-значения и не могут применяться в левой части операции присваивания. Поэтому в С был добавлен термин *модифицируемое l-значение* для идентификации объекта, чье значение может изменяться. Таким образом, левая часть оператора присваивания должна быть модифицируемым l-значением.

В современном стандарте предлагается более точный термин: *значение локатора объекта*.

Термин *r-значение* относится к величинам, которые могут быть присвоены модифицируемым l-значениям, но которые сами не являются l-значениями. Например, рассмотрим следующий оператор:

```
bmw = 2002;
```

Здесь bmw — это модифицируемое l-значение, а 2002 — r-значение. Как вы уже, вероятно, догадались, "r" в r-значении происходит от слова "right" — "правая часть"). R-значениями могут быть константы, переменные или любое другое выражение, которое в результате дает значение, например, вызов функции. И действительно, в современном стандарте вместо термина *r-значение* применяется термин *значение выражения*.

Ниже приведен небольшой пример:

```
int ex;
int why;
int zee;
const int TWO = 2;
why = 42;
zee = why;
ex = TWO * (why + zee);
```

В этом примере ex, why и zee — модифицируемые l-значения (или значения локаторов объектов). Они могут использоваться слева или справа от символа операции присваивания. Здесь TWO — не модифицируемое l-значение; оно может указываться только в правой части. (В контексте установки TWO в 2 операция = представляет инициализацию, а не присваивание, поэтому правило не нарушается.) В то же время 42 — это r-значение. Оно не ссылается на какую-то конкретную ячейку памяти. Кроме того, хотя why и zee — модифицируемые l-значения, выражение (why + zee) является r-значением. Оно не представляет конкретную ячейку памяти и ему нельзя присваивать значение. Это всего лишь временное значение, которое программа вычисляет, а затем отбрасывает по завершении работы с ним.

По мере ознакомления с этими понятиями вырисовывается подходящий термин для того, что мы называем "элементом" (примером может служить фраза "элемент слева от знака ="); таким термином является *операнд*. Операнды — это то, чем оперируют операции. Например, процесс поедания гамбургера можно описать применением операции "поедание" к операнду "гамбургер". Аналогично можно сказать, что левым операндом операции = должно быть модифицируемое l-значение.

Базовая операция присваивания в языке С несколько отличается от других операций. Рассмотрим короткую программу, показанную в листинге 5.3.

Листинг 5.3. Программа golf.c

```
/* golf.c -- таблица результатов турнира по гольфу */
#include <stdio.h>
int main(void)
{
   int jane, tarzan, cheeta;
```

```
    cheeta = tarzan = jane = 68;
    printf("                  чита    тарзан    джейн\n");
    printf("Счет первого раунда %4d %8d %8d\n",cheeta,tarzan,jane);

    return 0;
}
```

Многие языки программирования не разрешают тройное присваивание значений, сделанного в этой программе, но в С это считается обычным делом. Присваивание выполняется справа налево. Вначале значение 68 получает переменная jane, затем tarzan и, наконец, это значение присваивается переменной cheeta. В результате получается следующий вывод:

```
                      чита    тарзан    джейн
Счет первого раунда    68       68        68
```

Операция сложения: +

Операция сложения приводит к суммированию двух значений с обеих сторон знака +. Например, оператор

```
printf("%d", 4 + 20);
```

выводит число 24, но не выражение

```
4 + 20
```

Суммируемые значения (операнды) могут быть как переменными, так и константами. Таким образом, следующий оператор заставляет компьютер извлечь значения двух переменных, указанных в правой части оператора, выполнить их сложение, а результат сложения присвоить переменной income:

```
income = salary + bribes;
```

Напомним еще раз, что income, salary и bribes — это модифицируемые l-значения, поскольку каждое из них идентифицирует объект данных, которому может быть присвоено значение, но выражение salary + bribes является r-значением, т.е. вычисленным значением, не идентифицируемым конкретной областью памяти.

Операция вычитания: –

Операция вычитания вызывает вычитание числа, следующего за знаком -, из числа, находящегося перед этим знаком. Например, приведенный ниже оператор присваивает переменной takehome значение 200.0:

```
takehome = 224.00 - 24.00;
```

Операции + и - называются *бинарными*, или *двухместными*, т.е. они требуют указания *двух* операндов.

Операции знака: – и +

Знак "минус" может использоваться для указания или изменения алгебраического знака значения. Например, следующие операторы приводят к присваиванию переменной smokey значения 12:

```
rocky = -12;
smokey = -rocky;
```

Когда знак "минус" применяется подобным образом, он называется *унарной операцией*, которая выполняется над одним операндом (рис. 5.2).

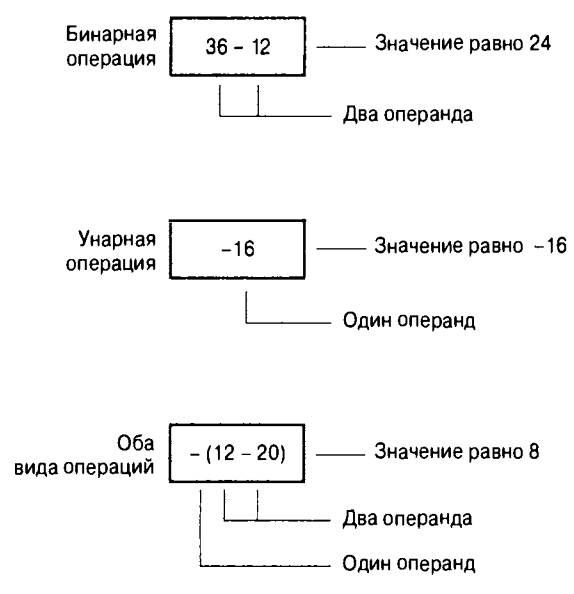


Рис. 5.2. Унарные и бинарные операции

Стандарт С90 вводит в язык С унарную операцию +. Она не меняет значения или знака операнда, но просто позволяет использовать такие операторы, как

```
dozen = +12;
```

и при этом не получать сообщений об ошибке. Раньше такая конструкция не допускалась.

Операция умножения: *

Умножение обозначается символом *. Например, следующий оператор умножает значение переменной inch на 2.54 и присваивает результат умножения переменной cm:

```
cm = 2.54 * inch;
```

Кстати, не желаете ли составить таблицу квадратов значений? В языке С отсутствует функция возведения в квадрат, но как показано в листинге 5.4, для вычисления квадратов значений можно применять операцию умножения.

Листинг 5.4. Программа squares.c

```
/* squares.c -- генерирует таблицу квадратов 20 значений */
#include <stdio.h>
int main(void)
{
   int num = 1;
   while (num < 21)
   {
      printf("%4d %6d\n", num, num * num);
      num = num + 1;
   }
   return 0;
}
```

Эта программа выводит 20 целых чисел и их квадраты, в чем вы можете убедиться самостоятельно. Давайте рассмотрим более интересный пример.

Экспоненциальный рост

Вы, скорее всего, слышали историю о могущественном правителе, который хотел вознаградить мудреца, оказавшего ему большую услугу. Когда мудреца спросили, что он желает получить, он указал на шахматную доску и попросил положить одно пшеничное зернышко на первую клетку, два зернышка на вторую клетку, четыре – на третью, восемь – на четвертую и т.д. Правитель, не имеющий понятия о математике, был поражен скромностью притязаний мудреца, поскольку был готов предложить ему большие богатства. Мудрец сыграл с правителем злую шутку, как показывает программа в листинге 5.5. Она вычисляет, сколько зернышек приходится на каждую клетку и подсчитывает общую сумму. Так как вы вряд ли следите за ежегодными объемами собранного урожая пшеницы, программа сравнивает промежуточные суммы с довольно приближенным суммарным значением годового урожая, собираемого во всем мире.

Листинг 5.5. Программа `wheat.c`

```
/* wheat.c -- экспоненциальный рост */
#include <stdio.h>
#define SQUARES 64          // количество клеток шахматной доски
int main(void)
{
   const double CROP = 2E16;    // мировой урожай пшеницы в зернах
   double current, total;
   int count = 1;
   printf("квадрат   добавлено  итого     ");
   printf("процент от \n");
   printf("         зерен       зерен    ");
   printf("мирового урожая\n");
        total = current = 1.0;  /* начинаем с одного зернышка */
   printf("%4d %13.2e %12.2e %12.2e\n", count, current,
   total, total/CROP);
   while (count < SQUARES)
   {
      count = count + 1;
      current = 2.0 * current;
      /* удвоить количество зерен на следующей клетке */
      total = total + current;   /* обновить итоговую сумму */
      printf("%4d %13.2e %12.2e %12.2e\n", count, current, total, total/CROP);
   }
   printf("Вот и все.\
   return 0;
}
```

Сначала выходные данные не должны были вызывать у правителя беспокойство:

```
клетка  добавлено    итого       процент от
          зерен      зерен     мирового урожая
   1    1.00e+00   1.00e+00      5.00e-17
   2    2.00e+00   3.00e+00      1.50e-16
   3    4.00e+00   7.00e+00      3.50e-16
   4    8.00e+00   1.50e+01      7.50e-16
   5    1.60e+01   3.10e+01      1.55e-15
   6    3.20e+01   6.30e+01      3.15e-15
   7    6.40e+01   1.27e+02      6.35e-15
   8    1.28e+02   2.55e+02      1.27e-14
   9    2.56e+02   5.11e+02      2.55e-14
  10    5.12e+02   1.02e+03      5.12e-14
```

На десяти клетках мудрец получил чуть больше тысячи зерен пшеницы, но взгляните на результат для 55 клетки:

```
55    1.80e+16    3.60e+16       1.80e+00
```

Плата мудреца превысила весь мировой урожай! Если хотите выяснить, что произойдет к 64-й клетке, выполните программу самостоятельно.

Приведенный пример иллюстрирует феномен экспоненциального роста. Население мира и расходование энергетических ресурсов растет по тому же закону.

Операция деления: /

В языке C символ / используется для обозначения деления. Значение, находящееся слева от символа /, делится на значение, указанное справа. Например, следующий оператор присваивает переменной `four` значение `4.0`:

```
four = 12.0/3.0;
```

Деление работает по-разному для целочисленных типов и типов с плавающей запятой. В результате деления с плавающей запятой получается число с плавающей запятой, а целочисленное деление дает целое число. Так как целое число не может иметь дробной части, деление 5 на 3 не является точным, поскольку результат не содержит дробной части. В языке C любая дробная часть, полученная при делении двух целых чисел, отбрасывается. Этот процесс называется *усечением.*

Запустите программу, показанную в листинге 5.6, чтобы посмотреть, как работает усечение, и узнать, чем отличается деление целых чисел от деления чисел с плавающей запятой.

Листинг 5.6. Программа divide.c

```
/* divide.c -- деление, каким мы его знаем */
#include <stdio.h>
int main(void)
{
   printf("Целочисленное деление: 5/4 равно %d \n", 5/4);
   printf("Целочисленное деление: 6/3 равно %d \n", 6/3);
   printf("Целочисленное деление: 7/4 равно %d \n", 7/4);
   printf("Деление с плавающей запятой: 7./4. равно %1.2f \n", 7./4.);
   printf("Смешанное деление: 7./4 равно %1.2f \n", 7./4);
   return 0;
}
```

В листинге 5.6 представлен случай "смешанных типов", когда значение с плавающей запятой делится на целое число. По сравнению с рядом других языков C более либерален и разрешает выполнять такие операции, однако обычно вы должны избегать смешивания типов. Ниже приведены результаты выполнения программы:

```
Целочисленное деление: 5/4 равно 1
Целочисленное деление: 6/3 равно 2
Целочисленное деление: 7/4 равно 1
Деление с плавающей запятой: 7./4. равно 1.75
Смешанное деление: 7./4 равно 1.75
```

Обратите внимание, что целочисленное деление не округляет до ближайшего целого, а всегда выполняет усечение (т.е. отбрасывает дробную часть). При смешивании в одной операции целых чисел и чисел с плавающей запятой ответ оказывается таким же, как и в случае деления чисел с плавающей запятой. На самом деле компьютер не

способен делить число с плавающей запятой на целое число, и поэтому компилятор преобразует оба операнда к одному типу. В данном случае перед выполнением деления целое число преобразуется в число с плавающей запятой.

До появления стандарта C99 язык C предоставлял разработчикам реализаций некоторую свободу в решении того, как должно выполняться деление отрицательных чисел. Одна из точек зрения заключалась в том, что процедура округления предусматривает нахождение наибольшего целого значения, которое меньше или равно числу с плавающей запятой. Естественно, число 3 удовлетворяет этому требованию, если его сравнивать с 3.8. Но как быть в случае -3.8? Метод нахождения наибольшего целого числа предполагает его округление до -4, поскольку -4 меньше, чем -3.8. Однако другая точка зрения на процесс округления состояла в том, что дробная часть просто отбрасывается; при такой интерпретации, называемой *усечением в направлении нуля*, предполагается преобразование числа -3.8 в -3. До выхода стандарта C99 в одних реализациях применялся первый подход, а в других — второй. Но в стандарте C99 определено усечение в направлении нуля, следовательно, -3.8 преобразуется в -3.

Свойства целочисленного деления оказываются очень удобными для решения некоторых задач, и вскоре вы ознакомитесь с примером. Сначала необходимо выяснить еще один важный аспект: что произойдет, если объединить несколько операций в один оператор? Это и является следующей темой.

Приоритеты операций

Рассмотрим следующую строку кода:

```
butter = 25.0 + 60.0 * n / SCALE;
```

В этом операторе присутствуют операции сложения, умножения и деления. Какая из них выполнится первой? Будет ли 25.0 суммироваться с 60.0, полученный результат 85.0 умножаться на n, после чего произведение делиться на SCALE? Или же 60.0 умножится на n, к полученному произведению прибавится 25.0, после чего результат сложения разделится на SCALE? А, может быть, будет использоваться вообще другой порядок выполнения операций? Предположим, что n равно 6.0, а SCALE — 2.0. При таких значениях первый подход даст результат 255, а второй — 192.5. По всей видимости, в программе на C задействован другой порядок, поскольку переменная butter в итоге получает значение 205.0.

Очевидно, что порядок выполнения различных операций оказывает влияние на конечный результат, следовательно, язык C нуждается в однозначных правилах выбора того, что должно выполняться в первую очередь. Эта проблема решается путем установки порядка выбора операций. Каждой операции назначается уровень *приоритета*. Как и в обычной арифметике, умножение и деление имеют более высокий приоритет, чем сложение и вычитание, поэтому они выполняются первыми. А что, если две операции обладают одинаковыми приоритетами? Когда они применяются к одному и тому же операнду, то выполняются в порядке следования внутри оператора. Большинство операций выполняются слева направо. (Операция = является исключением из этого правила.) Поэтому в операторе

```
butter = 25.0 + 60.0 * n / SCALE;
```

операции выполняются в следующем порядке:

```
60.0 * n
```

Сначала выполняется первая операция * или / в выражении (при условии, что n равно 6, 60.0 * n дает 360.0).

```
360.0 / SCALE
```

Затем выполняется вторая операция * или / в выражении.

```
25.0 + 180
```

В завершение (поскольку SCALE равно 2.0) выполняется первая операция + в выражении, давая результат 205.0.

Многие люди предпочитают представлять порядок вычислений с помощью диаграммы, которая называется *деревом выражения*. Пример такой диаграммы продемонстрирован на рис. 5.3. Диаграмма показывает, как исходное выражение постепенно сводится к одному значению.

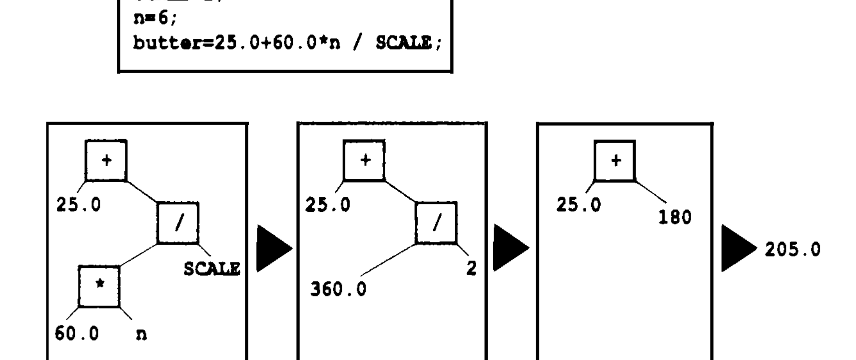


Рис. 5.3. Деревья выражений, показывающие операции, операнды и порядок вычисления

А что, если нужно обеспечить выполнение операции сложения раньше операции деления? Тогда можно поступить так, как это сделано в следующем операторе:

```
flour = (25.0 + 60.0 * n) / SCALE;
```

Все, что заключено в круглые скобки, выполняется первым. Внутри круглых скобок действуют обычные правила. В приведенном примере сначала выполняется умножение, а затем сложение. На этом вычисление выражения в круглых скобках завершено. Затем результат может быть разделен на SCALE.

Правила для рассмотренных операций обобщены в табл. 5.1.

Таблица 5.1. Операции в порядке снижения приоритета

Операции	Ассоциативность
()	Слева направо
+ - (унарные)	Справа налево
* /	Слева направо
+ - (бинарные)	Слева направо
=	Справа налево

Обратите внимание, что две формы использования знака "минус" имеют разные приоритеты, как и две формы применения знака "плюс". В столбце "Ассоциативность" указано, как операция связывается с ее операндами. Например, унарная операция - ассоциируется с величиной, находящейся справа от нее, а в случае операции деления операнд слева делится на операнд справа.

Приоритет и порядок вычисления

Приоритеты операций являются жизненно важным правилом для определения порядка вычисления выражения, но они не обязательно определяют полный порядок вычислений. Язык C оставляет определенную свободу выбора при реализации. Рассмотрим следующий оператор:

```
y = 6 * 12 + 5 * 20;
```

Приоритеты диктуют порядок вычисления, когда две операции применяются к одному операнду. Например, 12 является операндом для двух операций, * и +, и приоритет указывает, что умножение должно выполняться первым. Аналогично, согласно приоритетам, операнд 5 должен сначала умножаться, а не суммироваться. Короче говоря, операции умножения 6 * 12 и 5 * 20 выполняются до того, как начнет выполняться операция сложения. При этом приоритеты не устанавливают, какая из двух операций умножения выполняется первой. Язык C оставляет этот выбор за реализацией, поскольку один выбор может оказаться более эффективным для одного оборудования, а другой выбор обеспечивает более высокую производительность на другом оборудовании. В любом случае выражение сводится к 72 + 100, так что последовательность выполнения операций умножения в этом конкретном примере не оказывает влияния на окончательный результат. Однако у вас может возникнуть вопрос: поскольку умножение выполняется слева направо, не означает ли это, что самое левое умножение выполнится первым? Правило ассоциативности применяется к операциям, которые *совместно используют* операнд. Например, в выражении 12 / 3 * 2 операции / и *, обладающие одинаковыми приоритетами, совместно используют операнд 3. Таким образом, в этом случае применяется правило "слева направо", и выражение сводится к 4 * 2, т.е. 8. (Выполнение справа налево привело бы к 12 / 6, или 2. В этом случае выбор имеет значение.) В предыдущем примере две операции * не имеют общего операнда, так что правило "слева направо" не применяется.

Исследование правил

Давайте испытаем эти правила в более сложном примере (листинг 5.7).

Листинг 5.7. Программа rules.c

```
/* rules.c -- проверка приоритетов */
#include <stdio.h>
int main(void)
{
    int top, score;

    top = score = -(2 + 5) * 6 + (4 + 3 * (2 + 3));
    printf("top = %d, score = %d\n", top, score);

    return 0;
}
```

Какое значение выведет эта программа? Сначала вычислите его вручную, а затем выполните программу или прочитайте приведенное ниже пояснение, чтобы проверить свой ответ.

Первым делом, вычислению в круглых скобках назначается наивысший приоритет. Какое выражение в скобках, -(2 + 5) * 6 или (4 + 3 * (2 + 3)), вычисляется первым, зависит, как мы только что обсудили, от конкретной реализации. В данном примере любой выбор приводит к получению одного и того же результата, поэтому предположим, что выражение слева вычисляется первым. Более высокий приоритет выраже-

ния в скобках означает, что в подвыражении -(2 + 5) * 6 сначала вычисляется сумма (2 + 5), которая равна 7. Далее к 7 применяется унарная операция минус, что дает -7. Теперь данное выражение принимает такой вид:

```
top = score = -7 * 6 + (4 + 3 * (2 + 3))
```

На следующем шаге необходимо вычислить 2 + 3. Выражение принимает вид

```
top = score = -7 * 6 + (4 + 3 * 5)
```

Поскольку приоритет операции * выше, чем у операции +, выражение сводится к

```
top = score = -7 * 6 + (4 + 15)
```

а затем и к

```
top = score = -7 * 6 + 19
```

Умножаем -7 на 6 и получаем следующее выражение:

```
top = score = -42 + 19
```

Выполнив сложение, получим

```
top = score = -23
```

Теперь переменной score присваивается значение -23, после чего и top получает значение -23. Вспомните, что операция = выполняется справа налево.

Некоторые дополнительные операции

В языке С имеется около 40 операций, и некоторые из них применяются гораздо чаще других. Те, с которыми вы уже ознакомились, относятся к наиболее распространенным операциям, но к этому списку можно добавить еще четыре очень полезных операции.

Операция sizeof и тип size_t

С операцией sizeof мы уже имели дело в главе 3. Вспомните, что она возвращает размер своего операнда в байтах. (Вспомните также, что байт в языке С определен как размер, используемый для типа char. В прошлом байт чаще всего состоял из 8 битов, но некоторые наборы символов использовали байты большего размера.) Операндом может быть конкретный объект данных, такой как имя переменной, либо им может быть тип. Если это тип, скажем, float, то операнд должен быть помещен в круглые скобки. В примере, представленном в листинге 5.8, показаны обе формы операндов.

Листинг 5.8. Программа sizeof.c

```
// sizeof.c -- использование операции sizeof
// в примере применяется определенный в стандарте c99 модификатор %z; если в вашей
// системе модификатор %zd не поддерживается, попробуйте вместо него %u или %lu
#include <stdio.h>
int main(void)
{
   int n = 0;
   size_t intsize;
   intsize = sizeof (int);
   printf("n = %d, n состоит из %zd байтов; все значения int имеют %zd байтов.\n",
         n, sizeof n, intsize );
   return 0;
}
```

В языке С указано, что операция sizeof возвращает значение типа size_t. Это целочисленный тип без знака, но не совершенно новый тип. Напротив, как вы можете вспомнить из предыдущей главы, он определен в терминах стандартных типов. В С имеется механизм typedef (обсуждаемый в главе 14), который позволяет создавать псевдонимы для существующих типов. Например, следующее определение делает real еще одним именем для типа double:

```
typedef double real;
```

Теперь можно объявить переменную типа real:

```
real deal;  // использование typedef
```

Компилятор обнаружит слово real, учтет, что оператор typedef сделал real псевдонимом для double, и создаст переменную deal как относящуюся к типу double. Аналогично, в системе заголовочных файлов С оператор typedef может использоваться для того, чтобы сделать size_t синонимом unsigned int в одной системе и unsigned long в другой. Таким образом, когда вы применяете тип size_t, компилятор подставит стандартный тип, предназначенный для вашей системы.

Стандарт С99 идет на шаг дальше и предлагает %zd в качестве спецификатора функции printf() для вывода значения size_t. Если спецификатор %zd в вашей системе не реализован, вместо него можно попробовать %u или %lu.

Операция деления по модулю: %

Операция деления по модулю применяется в целочисленной арифметике. Ее результатом является *остаток от деления* целого числа, стоящего слева от знака операции, на число, расположенное справа от него. Например, 13 % 5 (читается как "13 по модулю 5") дает в результате 3, поскольку 5 умещается в 13 дважды с остатком, равным 3. Не пытайтесь выполнять эту операцию над числами с плавающей запятой. Она просто не работает.

На первый взгляд эта операция может показаться экзотическим инструментом, предназначенным только для математиков, но по существу она очень удобна и полезна. Обычно она используется, чтобы помочь управлять ходом выполнения программы. Предположим, например, что вы работаете над программой подготовки счетов, предназначенной для начисления дополнительной платы каждый третий месяц. Для этого достаточно разделить номер месяца по модулю 3 (т.е. month % 3) и проверить, не равен ли результат 0. Если равен, программа включает дополнительную плату. Это станет более понятным после ознакомления с оператором if в главе 7.

В листинге 5.9 приведен еще один пример применения операции %. В нем также демонстрируется еще один способ использования цикла while.

Листинг 5.9. Программа min_sec.c

```
// min_sec.c -- переводит секунды в минуты и секунды
#include <stdio.h>
#define SEC_PER_MIN 60          // количество секунд в минуте
int main(void)
{
   int sec, min, left;

   printf("Перевод секунд в минуты и секунды!\n");
   printf("Введите количество секунд (<=0 для выхода):\n");
   scanf("%d", &sec);          // чтение количества секунд
   while (sec > 0)
   {
```

```
        min = sec / SEC_PER_MIN;   // усеченное количество минут
        left = sec % SEC_PER_MIN;  // количество секунд в остатке
        printf("%d секунд - это %d минут(ы) %d секунд.\n", sec,
              min, left);
        printf("Введите следующее значение (<=0 для выхода):\n");
        scanf("%d", &sec);
    }
    printf("Готово!\n");
    return 0;
}
```

Вот как выглядит пример вывода:

```
Перевод секунд в минуты и секунды!
Введите количество секунд (<=0 для выхода):
154
154 секунд - это 2 минут(ы) 34 секунд.
Введите следующее значение (<=0 для выхода):
567
567 секунд - это 9 минут(ы) 27 секунд.
Введите следующее значение (<=0 для выхода):
0
Готово!
```

В коде из листинга 5.2 для управления циклом while применяется счетчик. Как только значение счетчика превысит заданный размер, цикл завершается. Однако для загрузки новых значений переменной sec код в листинге 5.9 использует функцию scanf(). Цикл продолжается до тех пор, пока это значение положительно. Когда пользователь вводит ноль или отрицательное значение, цикл завершается. Важной особенностью программы в обоих случаях является то, что каждая итерация цикла обновляет значение проверяемой переменной.

Что произойдет, если будет введено отрицательное значение? До того, как в стандарте C99 было установлено для целочисленного деления правило "усечения в направлении нуля", существовало несколько возможностей. Но когда действует это правило, вы получаете отрицательный результат деления по модулю, если первый операнд отрицателен, и положительный результат во всех остальных случаях:

11 / 5	равно 2	и	11 % 5	равно 1
11 / -5	равно -2	и	11 % -2	равно 1
-11 / -5	равно 2	и	-11 % -5	равно -1
-11 / 5	равно -2	и	-11 % 5	равно -1

Если поведение в вашей системе оказывается другим, значит, она не поддерживает стандарт C99. В любом случае, стандарт фактически утверждает, что если a и b являются целочисленными, то вы можете вычислить a%b путем вычитания (a/b)*b из a. Например, значение -11%5 можно вычислить следующим образом:

```
-11 - (-11/5) * 5 = -11 -(-2)*5 = -11 -(-10) = -1
```

Операции инкремента и декремента: ++ и --

Операция инкремента решает простую задачу: она увеличивает (инкрементирует) значение своего операнда на 1. Существуют две разновидности этой операции. В первом случае символы ++ располагаются перед изменяемой переменной; это *префиксная форма.*

Во втором случае символы ++ следуют сразу за переменной; это *постфиксная форма.* Эти две формы отличаются друг от друга по моменту выполнения инкрементирования. Сначала мы объясним подобные черты этих форм, а затем обратимся к различиям. Короткий пример, представленный в листинге 5.10, демонстрирует работу операции инкремента.

Листинг 5.10. Программа add_one.c

```
/* add_one.c -- инкремент: префиксная и постфиксная формы */
#include <stdio.h>
int main(void)
{
   int ultra = 0, super = 0;
   while (super < 5)
   {
      super++;
      ++ultra;
      printf("super = %d, ultra = %d \n", super, ultra);
   }
   return 0;
}
```

Выполнение программы add_one.c генерирует следующий вывод:

```
super = 1, ultra = 1
super = 2, ultra = 2
super = 3, ultra = 3
super = 4, ultra = 4
super = 5, ultra = 5
```

Программа одновременно дважды просчитала до пяти. Тот же результат можно было бы получить, заменив две операции инкремента следующими операторами присваивания:

```
super = super + 1;
ultra = ultra + 1;
```

Это достаточно простые операторы. Зачем создавать еще одно сокращение, не говоря уже о двух? Одна из причин заключается в том, что компактная форма позволяет улучшить читабельность и упростить программы. Данные операции придают программам изящество и элегантность, радуя глаз. Например, часть программы shoes2.c из листинга 5.2 можно переписать так:

```
shoe = 3.0;
while (shoe < 18.5)
{
   foot = SCALE * size + ADJUST;
   printf("%10.1f %20.2f дюймов\n", shoe, foot);
   ++shoe;
}
```

Тем не менее, вы по-прежнему не до конца задействовали преимущества операции инкремента. Фрагмент программы можно еще больше сократить, как показано ниже:

```
shoe = 2.0;
while (++shoe < 18.5)
{
   foot = SCALE*shoe + ADJUST;
   printf("%10.1f %20.2f дюймов\n", shoe, foot);
}
```

Здесь процесс инкрементирования и сравнения из цикла while объединены в одно выражение. Конструкции такого типа настолько часто встречаются в программах на С, что заслуживают более пристального внимания.

Во-первых, как работает такая конструкция? Все довольно просто. Значение переменной shoe увеличивается на 1, а затем сравнивается с 18.5. Если оно меньше 18.5, то операторы, заключенные в фигурные скобки, выполняются один раз. Затем shoe снова увеличивается на 1 и цикл продолжается до тех пор, пока значение shoe не станет достаточно большим. Чтобы скомпенсировать инкремент переменной shoe, выполненный перед первым вычислением значения переменной foot, мы уменьшили начальное значение shoe с 3.0 до 2.0 (рис. 5.4).

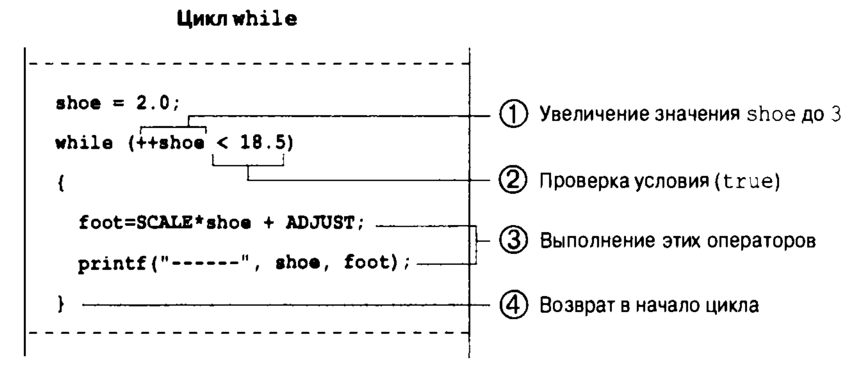


Рис. 5.4. Одна итерация цикла

Во-вторых, в чем преимущества такого подхода? Программа становится компактной. Но более важно то, что при этом объединяются в одном месте два процесса, которые управляют циклом. Первичным процессом является проверка того, продолжать ли выполнение цикла? В данном случае проверка заключается в выяснении, меньше ли значение переменной size, чем 18.5. Вторичный процесс изменяет значение проверяемого объекта; в данном случае размер обуви увеличивается на единицу.

Предположим, что вы забыли изменить размер обуви. Тогда значение переменной shoe *всегда* будет меньше 18.5, и цикл никогда не завершится. Вошедший в *бесконечный цикл* компьютер будет выводить одну и ту же строку. В конце концов, вы потеряете интерес к выводу и будете вынуждены как-то прервать выполнение программы. Проверка условия и изменение параметра цикла в одном месте, а не в разных, помогает помнить о необходимости изменить значение параметра цикла.

Недостаток объединения двух операций в одно выражение заключается в том, что это затрудняет понимание кода и увеличивает вероятность внесения ошибок.

Еще одно достоинство операции инкремента состоит в том, что она приводит к генерации несколько более эффективного кода на машинном языке по причине своего подобия действительным инструкциям машинного языка. Тем не менее, по мере улучшения компиляторов С поставщиками программного обеспечения это преимущество постепенно исчезает. Интеллектуальный компилятор способен распознать, что операцию x = x + 1 можно трактовать как ++x.

Наконец, эти операции обладают дополнительной характеристикой, которая может оказаться полезной в ряде тонких ситуаций. Чтобы выяснить, о чем идет речь, попробуйте запустить программу из листинга 5.11.

Листинг 5.11. Программа post_pre.c

```
/* post_pre.c -- постфиксная и префиксная формы */
#include <stdio.h>
int main(void)
{
    int a = 1, b = 1;
    int a_post, pre_b;

    a_post = a++;  // значение a++ во время этапа присваивания
    a_post = a++;  // значение ++b во время этапа присваивания
    printf("a  a_post  b  pre_b \n");
    printf("%1d %5d %5d %5d\n", a, a_post, b, pre_b);

    return 0;
}
```

Если все было сделано правильно, должен получиться следующий результат:

```
a  a_post  b  pre_b
2    1     2    2
```

Как и было задумано, значения переменных a и b увеличились на единицу. Однако a_post содержит значение переменной a *перед* изменением, а b_pre — значение переменной b *после* изменения. Именно в этом заключается отличие между префиксной и постфиксной формами операции инкремента (рис. 5.5):

```
a_post = a++;  // постфиксная форма: переменная a меняется после
               // использования ее значения
b_pre= ++b;    // префиксная форма: переменная b меняется до использования
               // ее значения
```

Когда одна из этих операций инкремента применяется сама по себе, как в одиночном операторе ego++;, ее форма не имеет значения. Однако выбранная форма играет роль, если операция и ее операнд являются частью более крупного выражения, как в только что показанных операторах присваивания. В ситуациях подобного рода нужно иметь четкое представление о том, какой результат вы желаете получить. В качестве примера вспомните, что мы намеревались использовать следующую конструкцию:

```
while (++shoe < 18.5)
```

Такая проверка условия завершения цикла позволяет получить таблицу значений вплоть до размера 18. Если вы укажете shoe++ вместо ++shoe, то таблица расширится до размера 19, т.к. значение shoe будет увеличиваться после сравнения, а не до него.

Конечно, вы могли бы возвратиться к менее элегантной форме:

```
shoe = shoe + 1;
```

но тогда никто не поверит, что вы настоящий программист на языке C.

При чтении этой книги вы должны уделять особое внимание примерам применения операции инкремента. Задайте себе вопрос, использовали вы префиксную и постфиксную формы взаимозаменяемо или же обстоятельства диктовали конкретный выбор. Возможно, более разумная политика предполагает отказ от кода, в котором есть разница, какая форма выбрана — префиксная или постфиксная. Например, вместо

```
b = ++i;  // если используется i++, значение переменной b будет другим
```

применяйте

```
++i;   // строка 1
b = i; // переменная b получит то же значение, даже если в строке 1 указать i++
```

Префиксная форма

```
q = 2*++a;
```

Сначала значение a увеличивается на 1, затем результат умножается на 2 и присваивается переменной q

Постфиксная форма

```
q = 2*a++;
```

Сначала значение a умножается на 2 и присваивается переменной q, после чего значение a увеличивается на 1

Рис. 5.5. Префиксная и постфиксная формы инкремента

Тем не менее, иногда интересно проявить некоторую беспечность, так что в данной книге не всегда соблюдается этот благоразумный совет.

Декрементирование: --

Для каждой формы операции инкремента имеется соответствующая форма *операции декремента.* Вместо ++ указывайте --:

```
--count;  // префиксная форма операции декремента
count--;  // постфиксная форма операции декремента
```

Листинг 5.12 служит иллюстрацией того, что компьютер может быть опытным поэтом.

Листинг 5.12. Программа bottles.c

```
#include <stdio.h>
#define MAX 100
int main(void)
{
   int count = MAX + 1;
   while (--count > 0) {
      printf("%d бутылок родниковой воды на полке, "
            "%d бутылок родниковой воды!\n", count, count);
      printf("Возьмите одну из них и пустите по кругу,\n");
      printf("%d бутылок родниковой воды!\n\n", count - 1);
   }
   return 0;
}
```

Вывод начинается примерно так:

```
100 бутылок родниковой воды на полке, 100 бутылок родниковой воды!
Возьмите одну из них и пустите по кругу,
99 бутылок минеральной воды!

99 бутылок родниковой воды на полке, 99 бутылок родниковой воды!
Возьмите одну из них и пустите по кругу,
98 бутылок родниковой воды!
```

Все это продолжается некоторое время и закачивается следующим образом:

```
1 бутылок родниковой воды на полке, 1 бутылок родниковой воды!
Возьмите одну из них и пустите по кругу,
0 бутылок родниковой воды!
```

Очевидно, что у опытного поэта есть проблема с множественным числом, но это можно исправить за счет использования условной операции, описанной в главе 7.

Кстати, операция > означает "больше чем". Как и < ("меньше чем"), она является *операцией отношения*. Более подробно операции отношений рассматриваются в главе 6.

Приоритеты операций

Операции инкремента и декремента имеют очень высокий приоритет; выше по приоритету только скобки. Поэтому `x*y++` означает `(x)*(y++)`, но не `(x*y)++`, что благоприятно, т.к. последняя конструкция не имеет смысла. Операции инкремента и декремента применяются только к *переменным* (или, в общем случае, к модифицируемым l-значениям), а само произведение `x*y` не является модифицируемым l-значением, хотя его части таковыми являются.

Не путайте приоритеты этих двух операций с порядком их вычисления. Предположим, мы имеем такой код:

```
y = 2;
n = 3;
nextnum = (y + n++)*6;
```

Какое значение получит `nextnum`? Подстановка значений дает следующее:

```
nextnum = (2 + 3)*6 = 5*6 = 30
```

Значение переменной `n` увеличивается до 4 только после ее использования. Приоритет операции говорит о том, что операция ++ применяется только к `n`, но не к `y + n`. Он также указывает, когда значение `n` используется для вычисления выражения, но момент изменения значения `n` определяется природой операции инкремента.

Когда `n++` является частью выражения, можно считать, что это означает "использовать переменную `n`, а затем увеличить ее значение на единицу". С другой стороны, `++n` означает "увеличить значение переменной `n` на единицу, а затем использовать ее".

Не умничайте

Слишком частое применение операции инкремента быстро приводит к путанице. Например, может показаться, что программу для вывода целых чисел и их квадратов `squares.c` (листинг 5.4) удастся улучшить, заменив в ней цикл `while` таким циклом:

```
while (num < 21)
{
   printf("%10d %10d\n", num, num*num++);
}
```

Код выглядит вполне разумно. Вы выводите число `num`, умножаете его само на себя, чтобы получить квадрат, а затем увеличиваете значение `num` на 1. Действительно, эта программа может даже работать в некоторых системах, правда, не во всех. Проблема заключается в том, что когда функция `printf()` собирается извлекать значения для вывода, она может сначала вычислить последний аргумент, увеличив значение `num` на 1, и только затем перейти к следующему аргументу.

В результате вместо того, чтобы вывести

```
5         25
```

она может вывести

```
6         25
```

Она даже может работать справа налево, используя 5 в качестве значения крайнего правого аргумента num и 6 в качестве значения следующих двух аргументов, приводя к такому выводу:

```
6        30
```

Компилятор языка С может самостоятельно решать, какой аргумент функции вычислять первым. Такая свобода выбора увеличивает эффективность компилятора, но может стать причиной проблем, если операция инкремента применяется к аргументу функции. Другим возможным источником неприятностей может стать оператор следующего вида:

```
ans = num/2 + 5*(1 + num++);
```

И снова проблема заключается в том, что компилятор может выполнять действия не в том порядке, который вы имели в виду. Вы могли предполагать, что сначала будет вычислено выражение num/2 и только затем произойдет переход к вычислению другой части выражения, однако компилятор вполне может первым вычислить последний элемент, увеличить значение num и использовать это новое значение для вычисления num/2. Здесь не существует никаких гарантий.

Еще одним источником проблем может стать такая конструкция:

```
n = 3;
y = n++ + n++;
```

Действительно, после выполнения второго оператора значение n увеличивается на 2, но значение y будет неопределенным. Компилятор может применить старое значение n дважды при вычислении значения y, а затем дважды увеличить значение n на единицу. При этом y принимает значение 6, а n — значение 5, либо он может использовать старое значение один раз, увеличить значение n один раз, задействовать это значение во втором экземпляре n в выражении, а затем инкрементировать n во второй раз. В таком случае y принимает значение 7, а n — значение 5. Возможен как первый, так и второй вариант. Точнее говоря, результат не определен, т.е. стандарт С не регламентирует, каким должен быть результат.

Описанных выше проблем легко избежать, если руководствоваться следующими рекомендациями.

- Не применяйте операцию инкремента или декремента к переменной, которая является частью более чем одного аргумента функции.
- Не используйте операцию инкремента или декремента с переменной, которая появляется в выражении более одного раза.

С другой стороны, в языке С имеются определенные гарантии относительно того, когда выполняется инкрементирование. Мы вернемся к этой теме, когда будем рассматривать точки следования в разделе "Побочные эффекты и точки следования" далее в главе.

Выражения и операторы

Мы применяли термины *выражение* и *оператор* на протяжении нескольких начальных глав, а теперь настало время подробнее рассмотреть их смысл. Операторы формируют базовые шаги программы на С, и большинство операторов построено на основе выражений. Отсюда следует, что в первую очередь необходимо подробно изучить выражения.

Выражения

Выражение состоит из комбинации операций и операндов. (Вспомните, что операнд – это то, над чем выполняется операция.) Простейшим выражением является отдельный операнд, и он может служить отправной точкой для построения более сложных выражений. Ниже приведено несколько примеров выражений:

```
4
-6
4+21
a*(b + c/d)/20
q = 5*2
x = ++q % 3
q > 3
```

Как видите, операндами могут быть константы, переменные и их сочетания. Некоторые выражения представляют собой комбинации выражений меньших размеров, называемых *подвыражениями*. Например, в четвертом примере c/d является подвыражением.

Каждое выражение имеет значение

Важное свойство языка С заключается в том, что каждое выражение имеет значение. Чтобы найти это значение, нужно выполнить операции в порядке, определенном приоритетами операций. Значения нескольких первых приведенных выше выражений очевидны, но что можно сказать о выражениях со знаком =? Эти выражения просто принимают те же значения, что и переменные, находящиеся слева от знака =. Поэтому выражение q=5*2 как единое целое получает значение 10. А что можно сказать о выражении q>3 ? Такие выражения отношения получают значение 1, если выражение истинно, и 0, если оно ложно. Ниже приведены несколько выражений и их значения.

Выражение	Значение
-4 + 6	2
c = 3 + 8	11
5 > 3	1
6 + (c = 3 + 8)	17

Последнее выражение выглядит странным. Тем не менее, оно вполне допустимо в С (но использовать выражения такого рода не рекомендуется), и представляет собой сумму двух подвыражений, каждое из которых имеет собственное значение.

Операторы

Операторы служат основными строительными блоками программы. *Программа* – это последовательность операторов с необходимыми знаками пунктуации. Оператор представляет собой завершенную инструкцию для компьютера. В языке С операторы распознаются по наличию точки с запятой в конце. Таким образом, код

```
legs = 4
```

является всего лишь выражением (которое может быть частью другого выражения), но

```
legs = 4;
```

становится оператором.

Простейшим из возможных считается пустой оператор:

```
; // пустой оператор
```

Он ничего не делает и является особым случаем инструкции.

Что же в целом формирует завершенную инструкцию? Прежде всего, в С любое выражение трактуется как оператор, если оно дополнено точкой с запятой. (Это называется *оператором выражения*.) Следовательно, в С не отклоняются строки, подобные показанным ниже:

```
8;
3 + 4;
```

Однако эти операторы ничего не делают в программе и в действительности не могут считаться осмысленными. Обычно операторы изменяют значения переменных и вызывают функции:

```
x = 25;
++x;
y = sqrt(x);
```

Хотя оператор (во всяком случае, осмысленный оператор) – это завершенная инструкция, не все завершенные инструкции являются операторами. Рассмотрим следующий оператор:

```
x = 6 + (y = 5);
```

В него входит подвыражение `y = 5`, представляющее завершенную инструкцию, но это только часть оператора. Поскольку завершенная инструкция не обязательно является оператором, для идентификации инструкций, которые действительно представляют собой операторы, необходима точка с запятой.

До сих пор мы уже встречались с пятью видами операторов (не считая пустого оператора). В листинге 5.13 показан короткий пример, в котором применяются все эти пять видов операторов.

Листинг 5.13. Программа addemup.c

```
/* addemup.c -- пять видов операторов */
#include <stdio.h>
int main(void)              /* находит сумму первых 20 целых чисел */
{
    int count, sum;         /* оператор объявления     */

    count = 0;              /* оператор присваивания   */
    sum = 0;                /* оператор присваивания   */
    while (count++ < 20)    /* оператор                */
        sum = sum + count;  /*     while               */
    printf("sum = %d\n", sum); /* оператор вызова функции */

    return 0;               /* оператор возврата       */
```

Давайте обсудим код в листинге 5.13. К этому моменту вы должны быть хорошо знакомы с оператором объявления. Тем не менее, напоминаем, что он устанавливает имена и типы переменных и выделяет для них пространство в памяти. Обратите внимание, что оператор объявления не является оператором выражения. То есть удаление символа точки с запятой ведет к чему-то, что не является выражением и не имеет значения:

```
int port                   /* это не выражение, оно не имеет значения */
```

Оператор присваивания – это "рабочая лошадка" многих программ; он присваивает значение переменной. Данный оператор состоит из имени переменной, за которым следует операция присваивания (=), а за ней – выражение, сопровождаемое точкой с запятой. Обратите внимание, что оператор `while` в примере содержит в себе оператор присваивания. Оператор присваивания представляет собой пример оператора выражения.

Оператор функции заставляет функцию делать то, для чего она предназначена. В рассматриваемом примере функция `printf()` вызывается для того, чтобы вывести некоторые результаты. Оператор `while` имеет три разных части (рис. 5.6). Первой частью является ключевое слово `while`. Вторая часть – это условие проверки, помещенное в круглые скобки. К третьей части относится оператор, который выполняется, если условие истинно. В цикл включен только один оператор. Им может быть простой оператор, как в рассматриваемом примере (в таком случае фигурные скобки для его обозначения не нужны), либо составной оператор, как в некоторых приведенных выше примерах (в этом случае наличие фигурных скобок обязательно). Составные операторы будут обсуждаться позже.

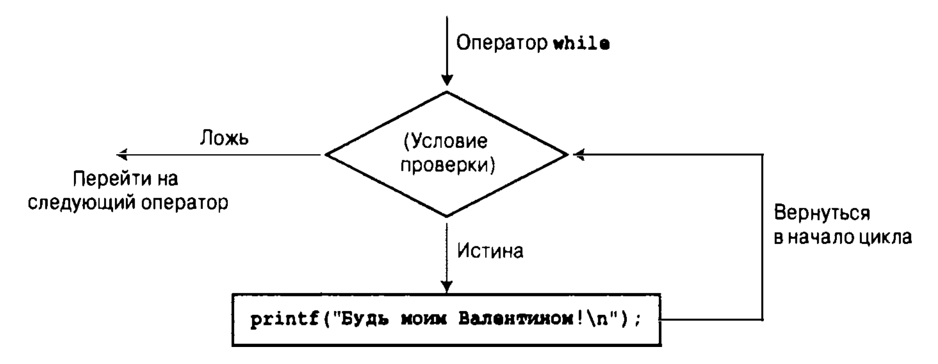


Рис. 5.6. *Структура простого цикла* `while`

Оператор `while` принадлежит к классу операторов, которые иногда называют *структурированными*, поскольку они обладают более сложной структурой, чем простой оператор присваивания. В последующих главах вы столкнетесь со многими другими структурированными операторами.

Оператор `return` завершает выполнение функции.

Побочные эффекты и точки следования

Введем еще несколько терминов языка С. *Побочный эффект* – это модификация объекта данных или файла. Например, побочный эффект оператора

```
states = 50;
```

заключается в том, что переменная `states` устанавливается в `50`. Но почему "побочный эффект"? Это больше похоже на основную цель оператора! Однако с точки зрения языка С основная цель состоит в вычислении выражений. Покажите С выражение `4 + 6`, и С вычислит его значение 10. Покажите выражение `states = 50`, и С вычислит его значение 50. С вычислением этого выражения связан побочный эффект, заключающийся в изменении значения переменной `states` на `50`. Операции инкремента и декремента, подобно операции присваивания, имеют свои побочные эффекты и используются главным образом из-за этих побочных эффектов.

Аналогично, при вызове функции printf() факт отображения ею информации считается побочным эффектом. (Вспомните, что значением функции printf() является количество отображенных элементов.)

Точка следования – это точка в ходе выполнения программы, в которой производится оценка всех побочных эффектов, прежде чем переходить на следующий шаг. В языке С точка следования помечается посредством точки с запятой в операторе. Это означает, что все изменения, вызванные операциями присваивания, инкремента и декремента в некотором операторе должны быть произведены до того, как управление в программе перейдет к следующему оператору. Некоторые операции, которые будут обсуждаться в следующих главах, имеют точки следования. Кроме того, конец любого полного выражения является точкой следования.

Что такое полное выражение? *Полное выражение* – это выражение, которое не является подвыражением более крупного выражения. Примерами полных выражений могут служить выражение в операторе присваивания, а также выражение, применяемое в условии проверки для цикла while.

Точки следования позволяют прояснить, когда происходит постфиксное инкрементирование. Рассмотрим в качестве примера следующий код:

```
while (guests++ < 10)
   printf("%d \n", guests);
```

Иногда начинающие программисты на С полагают, что фраза "использовать значение и затем инкрементировать его" означает в данном контексте увеличение значения переменной guests после ее использования в операторе printf(). Однако guests++ < 10 – это полное выражение, поскольку оно представляет собой условие проверки цикла while и, следовательно, конец этого выражения является точкой следования. Таким образом, язык С гарантирует, что побочный эффект (инкрементирование guests) произойдет до того, как программа перейдет к выполнению printf(). Тем не менее, применение постфиксной формы гарантирует то, что переменная guests будет инкрементирована после ее сравнения со значением 10.

Теперь рассмотрим следующий оператор:

```
y = (4 + x++) + (6 + x++);
```

Выражение 4 + x++ не является полным выражением, поэтому С не гарантирует, что значение x будет инкрементировано сразу после вычисления подвыражения 4 + x++. Здесь полное выражение представлено целым оператором присваивания, и точка с запятой отмечает точку следования. Таким образом, С может гарантировать только то, что к моменту перехода программы к выполнению следующего оператора значение x будет инкрементировано два раза. При этом в языке С не уточняется, будет ли значение x инкрементировано после вычисления каждого подвыражения или после вычисления всех выражений, что и является причиной, по которой следует избегать операторов подобного рода.

Составные операторы (блоки)

Составной оператор – это два или большее количество операторов, сгруппированных вместе путем помещения их в фигурные скобки; его также называют *блоком*. В программе shoes2.c блок используется для того, чтобы позволить оператору while содержать более одного оператора. Сравните следующие фрагменты кода:

```
/* фрагмент 1 */
index = 0;
```

```
while (index++ < 10)
   sam = 10 * index + 2;
printf("sam = %d\n", sam);

/* фрагмент 2 */
index = 0;
while (index++ < 10)
{
   sam = 10 * index + 2;
   printf("sam = %d\n", sam);
}
```

Внутри фрагмента 1 в цикл `while` включен только оператор присваивания. В отсутствие фигурных скобок область действия оператора `while` распространяется от ключевого слова `while` до следующей точки с запятой. Функция `printf()` вызывается только один раз – по завершении цикла.

Во фрагменте 2 наличие фигурных скобок гарантирует, что оба оператора являются частью цикла `while`, а функция `printf()` вызывается при каждом выполнении цикла. В терминах структуры оператора `while` весь составной оператор рассматривается как единственный оператор (рис. 5.7).

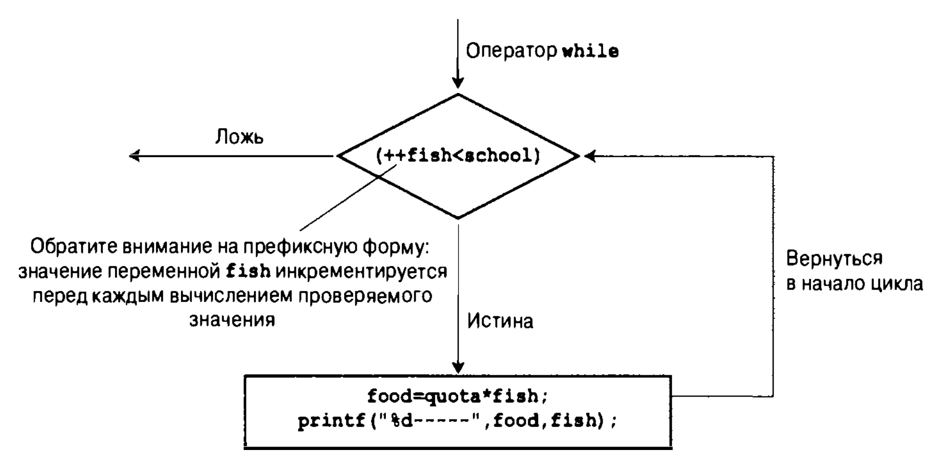


Рис. 5.7. Цикл `while` с составным оператором

СОВЕТ. Советы касательно стиля

Еще раз взгляните на оба фрагмента кода с операторами `while` и обратите внимание на пометку тела цикла с помощью отступов. Для компилятора отступы не имеют никакого значения. При интерпретации ваших инструкций он применяет фигурные скобки и свое знание структуры циклов `while`. Здесь отступы служат для того, чтобы облегчить зрительное восприятие организации программы.

Приведенный выше пример демонстрирует широко распространенный способ расстановки фигурных скобок в случае применения блока операторов, или составного оператора. Другой не менее распространенный способ выглядит следующим образом:

```
while (index++ < 10) {
   sam = 10*index + 2;
   printf("sam = %d \n", sam);
}
```

Этот стиль подчеркивает принадлежность блока к циклу `while`. В предыдущем стиле внимание акцентируется на том, что несколько операторов образуют блок. Повторимся еще раз: с точки зрения компилятора обе формы идентичны.

Подводя итоги, отметим: используйте отступы как инструмент, позволяющий сделать структуру программы более понятной для читателя.

Сводка: выражения и операторы

Выражения

Выражение представляет собой комбинацию операций и операндов. Простейшим выражением является константа или переменная без операции, например, `22` или `beebop`. Более сложные выражения выглядят подобно `55 + 22` и `vap = 2 * (vip + (vup = 4))`.

Операторы

Оператор — это инструкция компьютеру. Операторы бывают простыми и составными. *Простые операторы* завершаются точкой с запятой, как показано в следующих примерах.

Оператор объявления:	`int toes;`
Оператор присваивания:	`toes = 12;`
Оператор вызова функции:	`printf("%d\n", toes);`
Структурированный оператор:	`while (toes < 20) toes = toes + 2;`
Оператор возврата:	`return 0;`
Пустой оператор:	`; /* ничего не делает */`

Составные операторы, или *блоки*, состоят из одного или большего количества операторов (которые сами могут быть составными операторами), заключенных в фигурные скобки. Приведенный ниже пример оператора `while` содержит составной оператор:

```
while (years < 100)
{
   wisdom = wisdom * 1.05;
   printf("%d %d\n", years, wisdom);
   years = years + 1;
}
```

Преобразования типов

Операторы и выражения обычно должны использовать одни и те же типы выражений и констант. Однако если вы смешиваете типы, то язык С этому не препятствует, как это делает, скажем, Pascal. Взамен применяется набор правил, обеспечивающих автоматическое преобразование типов данных. Это может быть удобным, но также и опасным, особенно если смешивание типов происходит неумышленно. (Программа lint, входящая в состав многих систем Unix, проверяет наличие "конфликтов" между типами. Многие компиляторы С, которые ориентированы на системы, отличные от Unix, информируют о возможных проблемах с типами, если выбран высокий уровень сообщений об ошибках.) Неплохо иметь хотя бы общее представление о правилах преобразования типов.

Ниже описаны базовые правила преобразования типов данных.

1. Находясь в выражении, типы `char` и `short` (как `signed`, так и `unsigned`) автоматически преобразуются в `int` или при необходимости в `unsigned int`. (Если тип `short` имеет такой же размер, как у `int`, то размер типа `unsigned short` больше, чем `int`; в этом случае `unsigned short` преобразуется в `unsigned int`.) В K&R C, но не в текущей версии языка тип `float` автоматически преобразует-

ся в double. Поскольку они являются преобразованиями в большие по размеру типы, они называются *повышением*.

2. Если в любую операцию вовлечены два типа, оба значения приводятся к более высокому из этих двух типов.

3. Порядок типов от высшего к низшему выглядит так: long double, double, float, unsigned long long, long long, unsigned long, long, unsigned int и int. Возможно одно исключение, когда long и int имеют одинаковые размеры; в этом случае unsigned int превосходит long. Типы short и char в этом списке отсутствуют, т.к. они уже должны были повыситься до int или, возможно, до unsigned int.

4. В операторе присваивания финальный результат вычислений преобразуется к типу переменной, которой присваивается значение. Процесс может привести к повышению типа, как описано в правиле 1, или к *понижению* типа, при котором значение преобразуется в более низкий тип.

5. При передаче в качестве аргументов функции типы char и short преобразуются в int, а float — в double. Это автоматическое повышение переопределяется прототипированием функций, как будет показано в главе 9.

Повышение обычно представляет собой гладкий процесс без особых происшествий, но понижение может привести к реальной проблеме. Причина проста: типа более низкого уровня может оказаться недостаточно для сохранения полного числа. Например, 8-битная переменная char может хранить целочисленное значение 101, но не 22334. Что происходит, когда преобразованное значение не умещается в целевой тип? Ответ зависит от задействованных типов. Ниже приведены правила для случаев, когда присвоенное значение не помещается в конечном типе.

1. Когда целевым является одна из форм целочисленного типа без знака, а присвоенное значение представляет собой целое число, лишние биты, делающие значение слишком большим, игнорируются. Например, если целевой тип — 8-битный unsigned char, то присвоенным значением будет результат деления исходного значения по модулю 256.

2. Если целевым типом является целый тип со знаком, а присвоенное значение — целое число, то результат зависит от реализации.

3. Если целевой тип является целочисленным, а присвоенное значение представляет собой значение с плавающей запятой, то поведение не определено.

А что, если значение с плавающей запятой умещается в целочисленный тип? Когда типы с плавающей запятой понижаются до целочисленного типа, они усекаются или округляются в направлении нуля. Это означает, что и 23.12, и 23.99 усекаются до 23, а -23.5 усекается до -23.

В листинге 5.14 иллюстрируется работа некоторых из описанных правил.

Листинг 5.14. Программа convert.c

```
/* convert.c -- автоматическое преобразование типов */
#include <stdio.h>
int main(void)
{
    char ch;
    int i;
    float fl;
```

```
    fl = i = ch = 'C';                                    /* строка 9 */
    printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl);  /* строка 10 */
    ch = ch + 1;                                          /* строка 11 */
    i = fl + 2 * ch;                                      /* строка 12 */
    fl = 2.0 * ch + i;                                    /* строка 13 */
    printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl);  /* строка 14 */
    ch = 1107;                                            /* строка 15 */
    printf("Теперь ch = %c\n", ch);                       /* строка 16 */
    ch = 80.89;                                           /* строка 17 */
    printf("Теперь ch = %c\n", ch);                       /* строка 18 */

    return 0;
}
```

В результате выполнения программы convert.c получается следующий вывод:

```
ch = C, i = 67, fl = 67.00
ch = D, i = 203, fl = 339.00
Теперь ch = S
Теперь ch = P
```

Вот что происходит в системе, в которой реализованы 8-битный тип char и 32-битный тип int.

- Строки 9 и 10. Символ 'C' сохраняется как однобайтовое ASCII-значение в переменной ch. Целочисленная переменная i получает результат целочисленного преобразования символа 'C', который представляет собой число 67, сохраненное в 4 байтах памяти. И, наконец, переменная fl получает результат преобразования с плавающей запятой значения 67, которым является 67.00.
- Строки 11 и 14. Значение 'C' символьной переменной преобразуется в целое число 67, к которому затем добавляется 1. Результирующее 4-байтовое целое число 68 усекается до 1 байта и сохраняется в переменной ch. В случае вывода с использованием спецификатора %c число 68 интерпретируется как ASCII-код символа 'D'.
- Строки 12 и 14. При умножении на 2 значение переменной ch преобразуется в 4-байтовое целое (68). Результирующее целое значение (136) преобразуется в число с плавающей запятой, чтобы его можно было добавить к fl. Результат (203.00f) преобразуется в тип int и сохраняется в i.
- Строки 13 и 14. Значение переменной ch ('D', или 68) преобразуется в тип с плавающей запятой для его умножения на 2.0.Значение i (203) преобразуется в значение с плавающей запятой для выполнения сложения, и результат (339.00) сохраняется в переменной fl.
- Строки 15 и 16. Здесь предпринимается попытка понижения, когда переменной ch присваивается значение, выходящее за диапазон допустимых значений. После игнорирования лишних разрядов переменная ch в итоге получает значение, равное ASCII-коду символа 'S'. Точнее говоря, 1107 % 256 равно 83, что является кодом 'S'.
- Строки 17 и 18. Это еще один пример попытки понижения типа, при котором значение ch устанавливается равным числу с плавающей запятой. После усечения ch получает значение, равное ASCII-коду символа 'P'.

Операция приведения

Вы всегда должны избегать автоматического преобразования типов, особенно понижения, но иногда такие преобразования удобны при условии, что приняты все меры предосторожности. Преобразования типов, которые мы обсуждали до сих пор, выполняются автоматически. Тем не менее, можно потребовать выполнения нужного типа преобразований либо документировать факт, что вы знаете о выполнении преобразования типа. Такой метод называется *приведением* и предусматривает указание перед значением имени желаемого типа в круглых скобках. Скобки и имя типа вместе образуют *операцию приведения*. Вот общая форма операции приведения:

```
(тип)
```

Вместо слова *тип* указывается необходимый тип, например, `long`.

Рассмотрим две приведенные ниже строки кода, в которых `mice` – это переменная типа `int`. Вторая строка содержит два приведения к типу `int`.

```
mice = 1.6 + 1.7;
mice = (int) 1.6 + (int) 1.7;
```

В первой строке применяется автоматическое преобразование типов. Сначала суммируются числа `1.6` и `1.7`, что дает значение `3.3`. Затем это число преобразуется путем усечения в целое число 3, чтобы соответствовать переменной типа `int`. Во второй строке числа `1.6` и `1.7` перед сложением преобразуются в целочисленный вид (1), в результате чего переменной `mice` присваивается значение 1+1, или 2. По существу ни одна из форм не считается более корректной, чем другая; при выборе более подходящей необходимо учитывать контекст программируемой задачи.

Обычно вы не должны смешивать типы (именно поэтому некоторые языки не разрешают поступать так), но бывают случаи, когда смешивание полезно. Философия языка С направлена на устранение каких-либо препятствий на вашем пути и возлагает на вас ответственность за правильное употребление этой свободы.

Сводка: операции в С

Ниже перечислены операции, которые обсуждались выше.

Операция присваивания

= Присваивает переменной, указанной слева от знака операции, значение, заданное справа от него.

Арифметические операции

+ Добавляет значение справа от знака операции к значению слева от знака.

- Вычитает значение справа от знака операции из значения слева от знака.

- Как унарная операция, изменяет знак значения, указанного справа от знака операции.

* Умножает значение справа от знака операции на значение слева от знака.

/ Делит значение слева от знака операции на значение справа от знака. Если оба операнда являются целочисленными, результат усекается.

% Выдает остаток от деления значения слева от знака на значение справа от знака (только для целочисленных значений).

++ Добавляет 1 к значению переменной справа от знака операции (префиксная форма) или к значению слева от знака операции (постфиксная форма).

-- Подобна ++, но вычитает 1.

Операции различного назначения

sizeof Возвращает размер в байтах операнда, указанного справа. Операндом может быть спецификатор типа в круглых скобках, например, `sizeof (float)`, либо имя конкретной переменной, массива и т.д. без скобок, например, `sizeof foo`.

(*тип*) Как операция приведения, преобразует следующее за ней значение в тип, указанный внутри круглых скобок. Например, `(float) 9` преобразует целочисленное значение 9 в число с плавающей запятой `9.0`.

Функции с аргументами

К этому времени вы уже знакомы с использованием аргументов функций. Следующий шаг на пути к мастерству работы с функциями — научиться писать собственные функции, принимающие аргументы. Давайте кратко взглянем, в чем заключается этот навык. (Возможно, сейчас имеет смысл просмотреть еще раз пример функции `butler()`, приведенный в конце главы 2; он демонстрировал написание функции без аргументов.) Код в листинге 5.15 включает функцию `pound()`, которая выводит указанное количество знаков фунта (#). (Этот символ называют также знаком номера и хеш-символом.) Пример также иллюстрирует некоторые аспекты преобразования типов.

Листинг 5.15. Программа pound.c

```
/* pound.c -- определяет функцию с аргументом */
#include <stdio.h>
void pound(int n);          // объявление прототипа функции согласно стандарту ANSI
int main(void)
{
   int times = 5;
   char ch = '!';           // ASCII-код равен 33
   float f = 6.0f;
   pound(times);            // аргумент типа int
   pound(ch);               // эквивалентно pound((int)ch);
   pound(f);                // эквивалентно pound((int)f);
   return 0;
}
void pound(int n)           // заголовок функции в стиле ANSI, который указывает,
{                           // что функция принимает один аргумент int
   while (n-- > 0)
      printf("#");
   printf("\n");
}
```

Выполнение программы дает следующий вывод:

```
#####
#################################
######
```

Первым делом посмотрим на заголовок функции:

```
void pound(int n)
```

Если функция не принимает аргументов, в круглых скобках заголовка функции будет указано ключевое слово `void`. Поскольку функция `pound()` принимает один аргумент `int`, в круглых скобках содержится объявление переменной типа `int` по имени `n`. Здесь можно указывать любое имя, соответствующее правилам именования языка С.

Объявление аргумента создает переменную, которая называется *формальным аргументом* или *формальным параметром*. В этом случае формальным параметром является переменная int с именем n. Вызов функции, такой как pound(10), приводит к присваиванию переменной n значения 10. В программе вызов pound(times) присваивает n значение переменной times (5). Мы говорим, что вызов функции *передает* значение, которое называется *фактическим аргументом* или *фактическим параметром*, так что вызов функции pound(10) передает функции фактический аргумент 10, при этом значение 10 присваивается формальному параметру (переменной n). Другими словами, значение переменной times в функции main() копируется в новую переменную n внутри функции pound().

НА ЗАМЕТКУ! Аргументы или параметры

Хотя термины *аргумент* и *параметр* часто применяются взаимозаменяемо, в стандарте C99 решено использовать термин *аргумент* для фактического аргумента или фактического параметра и термин *параметр* для формального параметра и формального аргумента. С учетом этого соглашения можно сказать, что параметры — это переменные, а аргументы — это значения, которые предоставляются вызовом функции и присваиваются соответствующим параметрам. Таким образом, в листинге 5.15 аргументом функции pound() является times, а ее параметром — n. Подобным же образом в вызове функции pound(times+4) значение выражения times+4 представляет собой аргумент.

Имена переменных являются закрытыми внутри функции. Это значит, что имя, определенное в одной функции, не будет конфликтовать с таким же именем, объявленным в каком-то другом месте. Если бы в функции pound() вместо n использовалась переменная times, это привело бы к созданию переменной, отличной от times в функции main(). То есть вы получили бы две переменных с одним и тем же именем, но программе хорошо известно, к какой функции относится та или иная переменная.

Теперь взглянем на вызовы функций. Первым вызовом является pound(times) и, как уже было сказано, он приводит к присваиванию переменной n значения times, равного 5. В результате эта функция выводит пять знаков фунта и знак новой строки.

Второй вызов функции – pound(ch). В данном случае переменная ch имеет тип char. Она инициализируется символом !, который в системах с кодировкой ASCII имеет числовое значение 33. Но char является неподходящим типом для функции pound(). Именно здесь вступает в действие прототип функции, расположенный в верхней части программы. *Прототип* – это объявление функции, в котором дается описание возвращаемого значения функции и всех ее аргументов. Рассматриваемый прототип сообщает следующие сведения о функции pound():

- функция не возвращает никакого значения (часть void);
- функция принимает один аргумент, которым является значение типа int.

В этом случае прототип информирует компилятор о том, что функция pound() ожидает передачи ей аргумента типа int. В ответ компилятор, достигая выражения pound(ch), автоматически применяет к аргументу ch приведение типа, преобразуя его в аргумент int. В данной системе аргумент преобразуется из значения 33, хранящегося в одном байте, в значение 33, размещенное в четырех байтах, в результате чего значение 33 приобретает корректную форму, чтобы его можно было использовать в данной функции. Аналогично в последнем вызове, pound(f), приведение типа применяется для преобразования переменной f типа float в тип, подходящий этому аргументу.

До выхода стандарта ANSI C в языке использовались объявления функций, которые не были прототипами – они только указывали имя функции и тип возвращаемого значения, но не типы аргументов. В целях обратной совместимости в C по-прежнему допускается применение такой формы:

```
void pound();      /* объявление функции в стиле, предшествующем ANSI */
```

А что случится, если в программе pound.c вместо прототипа использовать такую форму объявления? Первый вызов функции, pound(times), будет работать, поскольку типом аргумента times является int. Второй вызов, pound(ch), также будет работать, т.к. при отсутствии прототипа компилятор C автоматически повышает типы аргументов char и short до int. Однако третий вызов, pound(f), оказывается неудачным, потому что в условиях отсутствия прототипа тип float автоматически повышается до double, а от этого, в действительности, мало пользы. Программа по-прежнему будет выполняться, но ее поведение окажется некорректным. Это можно было бы исправить, используя явное приведение типа в вызове функции:

```
pound ((int) f);  // принудительное использование нужного типа
```

Обратите внимание, что это может не помочь, если значение переменной f слишком велико, чтобы уместиться в тип int.

Демонстрационная программа

В листинге 5.16 представлена полезная программа (для физически активной части человечества), которая иллюстрирует несколько идей, рассматриваемых в данной главе. Она выглядит довольно длинной, однако все вычисления выполняются в шести строках кода, расположенных ближе к концу. Бальшая часть кода связана с обменом информацией между компьютером и пользователем. Код снабжен комментариями, позволяющими прояснить его работу. Просмотрите код, а затем мы обсудим несколько моментов.

Листинг 5.16. Программа running.c

```
#include <stdio.h>
const int S_PER_M = 60;               // количество секунд в минуте
const int S_PER_H = 3600;             // количество секунд в часе
const double M_PER_K = 0.62137;       // количество миль в километре
int main(void)
{
    double distk, distm;  // дистанция пробега в километрах и милях
    double rate;          // средняя скорость в милях в час
    int min, sec;         // время пробега в минутах и секундах
    int time;             // время пробега только в секундах
    double mtime;         // время пробега одной мили в секундах
    int mmin, msec;       // время пробега одной мили в минутах и секундах

    printf("Эта программа преобразует время пробега дистанции в метрической системе\n");
    printf("во время пробега одной мили и вычисляет вашу среднюю\n");
    printf("скорость в милях в час.\n");
    printf("Введите дистанцию пробега в километрах.\n");
    scanf("%lf", &distk); // %lf для типа double
    printf("Введите время в минутах и секундах.\n");
    printf("Начните с ввода минут.\n");
    scanf("%d", &min);
    printf("Теперь введите секунды.\n");
    scanf("%d", &sec);
```

```
    // переводит время в секунды
    time = S_PER_M * min + sec;
    // переводит километры в мили
    distm = M_PER_K * distk;
    // умножение миль в секунду на количество секунд в часе дает количество миль в час
    rate = distm / time * S_PER_H;
    // деление времени на расстояние дает время пробега одной мили
    mtime = (double) time / distm;
    mmin = (int) mtime / S_PER_M; // вычисление полного количества минут
    msec = (int) mtime % S_PER_M; // вычисление остатка в секундах
    printf("Вы пробежали %1.2f км (%1.2f мили) за %d мин, %d сек.\n",
           distk, distm, min, sec);
    printf("Такая скорость соответствует пробегу одной мили за %d мин, ",
           mmin);
    printf("%d сек.\nВаша средняя скорость составила %1.2f миль в секунду.\n",
           msec, rate);

    return 0;
}
```

В программе из листинга 5.16 применяется тот же подход, который использовался ранее в программе `min_sec.c` для преобразования финального времени в минуты и секунды, но здесь также выполняются преобразования типов. Почему? Причина в том, что для части программы, реализующей пересчет секунд в минуты, требуются целочисленные аргументы, а при преобразовании данных метрической системы в мили применяются числа с плавающей запятой. Чтобы сделать эти преобразования явными, мы использовали операцию приведения.

По правде говоря, существует возможность написания этой программы с применением только автоматических преобразований. На самом деле, мы так и поступили, объявив переменную `mtime` с типом `int`, что обеспечило принудительное преобразование результата вычисления времени в целочисленную форму. Однако данная версия программы отказалась работать на одной из 11 опробованных систем. Использованный компилятор (устаревшей и вышедшей из употребления версии) не смог следовать правилам языка С. Применение приведений типов делают ваши намерения более ясными не только для читателя кода, но, вполне вероятно, что также и для компилятора.

Ниже показан пример вывода:

```
Эта программа преобразует время пробега дистанции в метрической системе
во время пробега одной мили и вычисляет вашу среднюю
скорость в милях в час.
Введите дистанцию пробега в километрах.
10.0
Введите время в минутах и секундах.
Начните с ввода минут.
36
Теперь введите секунды.
23
Вы пробежали 10.00 км (6.21 мили) за 36 мин, 23 сек.
Такая скорость соответствует пробегу одной мили за 5 мин, 51 сек.
Ваша средняя скорость составила 10.25 миль в час
```

Ключевые понятия

Операции в языке С используются для предоставления разнообразных услуг. Каждую операцию можно характеризовать количеством требуемых операндов, ее

приоритетом и ассоциативностью. Два последних качества определяют, какая операция применяется первой, когда две операции совместно используют один операнд. Операции комбинируются со значениями для построения выражений, и с каждым выражением в C связано значение. Если не учитывать приоритет и ассоциативность операций, могут получиться выражения, которые не являются допустимыми или дают значения, отличные от ожидаемых; вряд ли это будет содействовать вашей репутации как программиста.

Язык C позволяет записывать выражения, объединяя разные числовые типы. Но арифметические операции требуют, чтобы операнды принадлежали одному и тому же типу, поэтому C выполняет автоматические преобразования. Тем не менее, рекомендуемая практика программирования – не полагаться на автоматические преобразования. Вместо этого делайте выбор типов явным, либо объявляя переменные необходимого типа, либо используя приведения типов. При таком подходе вам не придется сталкиваться с неожиданными автоматическими преобразованиями.

Резюме

В языке C существует много операций, таких как операции присваивания и арифметические операции, рассмотренные в этой главе. В общем случае *операция* выполняется над одним или большим количеством операндов с целью получения значения. Операции, которые принимают один операнд, вроде знака "минус" или `sizeof`, называются *унарными*. Операции, требующие два операнда, такие как операции сложения и умножения, называются *бинарными*.

Выражения – это комбинации операций и операндов. В языке C каждое выражение имеет значение, включая выражения присваивания и сравнения. Правила *приоритета операций* помогают определить, как группировать элементы при вычислении выражений. Когда две операции совместно используют один операнд, первой будет применена операция, имеющая более высокий приоритет. Если приоритеты операций равны, порядок их применения определяется ассоциативностью (слева направо или справа налево).

Операторы – это завершенные инструкции для компьютера. Они опознаются в C по завершающей точке с запятой. К этому моменту вы уже работали с операторами объявлений, операторами присваивания, операторами вызова функций и управляющими операторами. Операторы, заключенные в фигурные скобки, образуют *составной оператор*, или *блок*. Конкретным примером управляющего оператора является цикл `while`, который повторно выполняет операторы до тех пор, пока условие проверки остается истинным.

В языке C многие *преобразования типов* происходят автоматически. Типы `char` и `short` повышаются до типа `int` всякий раз, когда используются в выражениях или в качестве аргументов функции, не имеющей прототипа. Тип `float` в случае применения в аргументах функции повышается до типа `double`. В версии K&R C (но не в ANSI C) тип `float` повышается до `double`, если используется внутри выражения. Когда значение одного типа присваивается переменной второго типа, это значение преобразуется в тип, который имеет переменная. В случае преобразования больших типов в меньшие (например, `long` в `short` или `double` в `float`) возможна потеря данных. При смешивании типов в арифметических выражениях меньшие типы преобразуются к большим согласно правилам, изложенным в настоящей главе.

При определении функции, которая принимает аргумент, в определении функции вы объявляете *переменную*, или *формальный аргумент*. Затем значение, передаваемое в вызове функции, присваивается этой переменной и может применяться внутри данной функции.

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля приведены в приложении А.

1. Предположим, что все переменные имеют тип `int`. Найдите значение для каждой из следующих переменных:

   а. `x = (2 + 3) * 6;`

   б. `x = (12 + 6)/2*3;`

   в. `y = x = (2 + 3)/4;`

   г. `y = 3 + 2*(x = 7/2);`

2. Предположим, что все переменные имеют тип `int`. Найдите значение для каждой из следующих переменных:

   а. `x = (int) 3.8 + 3.3;`

   б. `x = (2 + 3) * 10.5;`

   в. `x = 3 / 5 * 22.0;`

   г. `x = 22.0 * 3 / 5;`

3. Вычислите каждое из следующих выражений:

   а. `30.0 / 4.0 * 5.0;`

   б. `30.0 / (4.0 * 5.0);`

   в. `30 / 4 * 5;`

   г. `30 * 5 / 4;`

   д. `30 / 4.0 * 5;`

   е. `30 / 4 * 5.0;`

4. Есть подозрение, что в приведенной ниже программе присутствуют ошибки. Сможете ли вы их обнаружить?

```
int main(void)
{
   int i = 1,
   float n;
   printf("Будьте внимательны! Далее идет последовательность дробей!\n");
   while (i < 30)
      n = 1/i;
      printf(" %f", n);
   printf("На этом все!\n");
   return;
}
```

5. Ниже приведен альтернативный вариант программы из листинга 5.9. Выглядит так, будто преследовалась цель упростить код путем замены двух операторов `scanf()` из листинга 5.9 единственным оператором `scanf()`. Почему этот вариант программы хуже исходного?

```
#include <stdio.h>
#define S_TO_M 60
int main(void)
{
   int sec, min, left;
```

```
    printf("Эта программа переводит секунды в минуты и ");
    printf("секунды.\n");
    printf("Введите количество секунд.\n");
    printf("Для завершения программы введите 0.\n");
    while (sec > 0) {
       scanf("%d", &sec);
       min = sec/S_TO_M;
       left = sec % S_TO_M;
       printf("%d секунд - это %d минут %d секунд. \n", sec, min, left);
       printf("Следующее значение?\n");
       }
    printf("До свидания!\n");
    return 0;
}
```

6. Что выведет эта программа?

```
#include <stdio.h>
#define FORMAT "%s! C is cool!\n"
int main(void)
{
    int num = 10;

    printf(FORMAT,FORMAT);
    printf("%d\n", ++num);
    printf("%d\n", num++);
    printf("%d\n", num--);
    printf("%d\n", num);
    return 0;
}
```

7. Что выведет следующая программа?

```
#include <stdio.h>
int main(void)
{
    char c1, c2;
    int diff;
    float num;

    c1 = 'S';
    c2 = 'O';
    diff = c1 - c2;
    num = diff;
    printf("%c%c%c:%d %3.2f\n", c1, c2, c1, diff, num);
    return 0;
}
```

8. Что выведет эта программа?

```
#include <stdio.h>
#define TEN 10
int main(void)
{
    int n = 0;

    while (n++ < TEN)
       printf("%5d", n);
    printf("\n");
    return 0;
}
```

9. Модифицируйте последнюю программу так, чтобы вместо чисел она выводила буквы алфавита от a до g.

10. Если бы следующие фрагменты кода были частью завершенной программы, тогда что они выводили бы?

а.
```
int x = 0;

while (++x < 3)
   printf("%4d", x);
```

б.
```
int x = 100;

while (x++ < 103)
printf("%4d\n",x);
   printf("%4d\n",x);
```

в.
```
char ch = 's';

while (ch < 'w')
{
   printf("%c", ch);
   ch++;
}
printf("%c\n",ch);
```

11. Что выведет следующая программа?

```
#define MESG "COMPUTER BYTES DOG"
#include <stdio.h>
int main(void)
{
    int n = 0;

    while ( n < 5 )
       printf("%s\n", MESG);
       n++;
    printf("На этом все.\n");
    return 0;
}
```

12. Напишите операторы, которые выполняют перечисленные ниже действия (или, другими словами, имеют следующие побочные эффекты).
   а. Увеличивает значение переменной x на 10.
   б. Увеличивает значение переменной x на 1.
   в. Присваивает переменной c удвоенную сумму a и b.
   г. Присваивает переменной c сумму a и удвоенного значения b.

13. Напишите операторы, которые выполняют перечисленные ниже действия.
   а. Уменьшает значение переменной x на 1.
   б. Присваивает m остаток от деления n на k.
   в. Делит q на b минус a и присваивает результат p.
   г. Присваивает переменной x результат деления суммы a и b на произведение c и d.

Упражнения по программированию

1. Напишите программу, которая преобразует время в минутах в часы и минуты. Для значения 60 создайте символическую константу посредством `#define` или `const`. Используйте цикл `while`, чтобы обеспечить пользователю возможность повторного ввода значений и для прекращения цикла, если вводится значение времени, меньшее или равное нулю.

2. Напишите программу, которая запрашивает у пользователя ввод целого числа, а затем выводит все целые числа, начиная с этого числа (и включая его) и заканчивая числом, которое больше введенного значения на 10 (включая его). (То есть, если вводится число 5, то в выводе должны присутствовать числа от 5 до 15.) Обеспечьте разделение выводимых значений друг от друга пробелами, символами табуляции или символами новой строки.

3. Напишите программу, которая запрашивает у пользователя ввод количества дней и затем преобразует это значение в количество недель и дней. Например, 18 дней программа должна преобразовать в 2 недели и 4 дня. Отображайте результаты в следующем формате:

   ```
   18 дней составляют 2 недели и 4 дня.
   ```

   Чтобы пользователь мог многократно вводить количество дней, используйте цикл `while`. Цикл должен завершаться при вводе пользователем неположительного значения, например, `0` или `-20`.

4. Напишите программу, которая запрашивает у пользователя ввод значения высоты в сантиметрах, после чего отображает высоту в сантиметрах, а также в футах и дюймах. Должны быть разрешены дробные части сантиметров и дюймов. Программа должна позволить пользователю продолжать ввод значений высоты до тех пор, пока не будет введено неположительное значение. Вывод этой программы должен иметь следующий вид:

   ```
   Введите высоту в сантиметрах: 182
   182.0 см = 5 футов, 11.7 дюймов
   Введите высоту в сантиметрах (<=0 для выхода из программы): 168.7
   168.0 см = 5 футов, 6.4 дюймов
   Введите высоту в сантиметрах (<=0 для выхода из программы): 0
   Работа завершена.
   ```

5. Внесите изменения в программу `addemup.c` (листинг 5.13), которая вычисляет сумму первых 20 целых чисел. (Если хотите, можете считать `addemup.c` программой, которая вычисляет сумму, которую вы будете иметь спустя 20 дней, если в первый день вы получаете $1, во второй день — $2, в третий день — $3 и т.д.) Модифицируйте программу так, чтобы можно было интерактивно указать, насколько далеко должно распространяться вычисление. Другими словами, замените число `20` переменной, значение которой вводится пользователем.

6. Теперь модифицируйте программу из упражнения 5, чтобы она вычисляла сумму квадратов целых чисел. (Или, если вам так больше нравится, программа должна вычислять сумму, которую вы получите, если в первый день вам заплатят $1, во второй день — $4, в третий день — $9 и т.д.) В языке C отсутствует функция возведения в квадрат, но, как вы знаете, квадрат числа `n` равен `n*n`.

7. Напишите программу, которая запрашивает ввод числа типа double и выводит значение куба этого числа. Для этого используйте собственную функцию, которая возводит значение в куб и выводит полученный результат. Программа main() должна передавать этой функции вводимое значение.

8. Напишите программу, которая выводит результаты применения операции деления по модулю. Пользователь должен первым ввести целочисленное значение, которое используется в качестве второго операнда и остается неизменным. Затем пользователь должен вводить числа, для которых будет вычисляться результат деления по модулю. Процесс должен прерываться вводом значения, которое равно или меньше 0. Пример выполнения этой программы должен выглядеть следующим образом:

```
Эта программа вычисляет результаты деления по модулю.
Введите целое число, которое будет служить вторым операндом: 256
Теперь введите первый операнд: 438
438 % 256 равно 182
Введите следующее число для первого операнда (<= 0 для выхода из
программы): 1234567
1234567 % 256 равно 135
Введите следующее число для первого операнда (<= 0 для выхода из
программы): 0
Готово
```

9. Напишите программу, которая запрашивает у пользователя ввод значения температуры по Фаренгейту. Программа должна считывать значение температуры как число типа double и передавать его в виде аргумента пользовательской функции по имени Temperatures(). Эта функция должна вычислять эквивалентные значения температуры по Цельсию и по Кельвину и отображать на экране все три значения температуры с точностью до двух позиций справа от десятичной точки. Функция должна идентифицировать каждое значение символом соответствующей температурной шкалы. Вот формула перевода температуры по Фаренгейту в температуру по Цельсию:

$$\textit{Температура по Цельсию} = 5.0 \;/\; 9.0 \times (\textit{температура по Фаренгейту} - 32.0)$$

В шкале Кельвина, которая обычно применяется в науке, 0 представляет абсолютный нуль, т.е. минимальный предел возможных температур. Формула перевода температуры по Цельсию в температуру по Фаренгейту имеет вид:

$$\textit{Температура по Кельвину} = \textit{температура по Цельсию} + 273.16$$

Функция Temperatures() должна использовать const для создания символических представлений трех констант, которые участвуют в преобразованиях. Чтобы предоставить пользователю возможность многократного ввода значений температуры, в функции main() должен быть организован цикл, который завершается при вводе символа q или другого нечислового значения. Воспользуйтесь тем фактом, что функция scanf() возвращает количество прочитанных ею элементов, поэтому она возвратит 1, если прочитает число, но не будет возвращать 1, когда пользователь введет q. Операция == выполняет проверку на равенство, так что ее можно применять для сравнения возвращаемого значения scanf() с 1.

6

Управляющие операторы C: циклы

В ЭТОЙ ГЛАВЕ...

- Ключевые слова:
  for
  while
  do while
- Операции:
  < > >=
  <= != == +=
  *= -= /= %=
- Функции:
  fabs()
- Три структуры циклов в C: *while*, *for* и *do while*
- Использование операций отношения для создания выражений, управляющих этими циклами
- Другие операции
- Массивы, которые часто используются с циклами
- Написание функций, имеющих возвращаемые значения

Мощный, интеллектуальный, универсальный и удобный! Несомненно, большинство из нас хотели бы заслужить такие эпитеты. Благодаря языку С, у наших программ появляется шанс получить подобную оценку. Сложность кроется в управлении ходом выполнения программы. Согласно теории вычислительных систем (это наука о компьютерах, но не наука, развивающаяся благодаря компьютерам,... во всяком случае, пока что), хороший язык программирования должен предоставлять следующие три формы потока управления программы.

- Выполнение последовательности операторов.
- Повторение последовательности операторов до тех пор, пока удовлетворяется некоторое условие (цикл).
- Использование проверки для выбора между альтернативными последовательностями операторов (условный переход).

Первая форма вам хорошо знакома; все приведенные ранее программы состояли из последовательностей операторов. Цикл while является одним из примеров второй формы. В этой главе мы более подробно рассмотрим цикл while , а также две других структуры циклов — for и do while. Третья форма, т.е. выбор между разными возможными последовательностями действий, делает программу намного "интеллектуальнее" и значительно увеличивает полезность компьютера как такового. К сожалению, вам придется дождаться следующей главы, прежде чем вы получите в свое распоряжение всю эту мощь. В настоящей главе также дано введение в массивы, поскольку именно к ним могут быть приложены новые знания о циклах. В дополнение в главе вы продолжите изучение функций. Давайте начнем с повторного обзора цикла while.

Повторный обзор цикла while

Вы уже кое-что знаете о цикле while, но мы повторим то, что известно, на примере программы, которая суммирует целые числа, вводимые с клавиатуры (листинг 6.1). В этом примере для определения момента прекращения ввода данных используется возвращаемое значение функции scanf().

Листинг 6.1. Программа summing.c

```
/* summing.c -- суммирует целые числа, вводимые в интерактивном режиме */
#include <stdio.h>
int main(void)
{
   long num;
   long sum = 0L;        /* инициализация переменной sum нулем */
   int status;

   printf("Введите целое число для последующего суммирования ");
   printf("(или q для завершения программы): ");
   status = scanf("%ld", &num);
   while (status == 1) /* == обозначает равенство */
   {
      sum = sum + num;
      printf("Введите следующее целое число (или q для завершения программы): ");
      status = scanf("%ld", &num);
   }
   printf("Сумма введенных целых чисел равна %ld.\n", sum);

   return 0;
}
```

В программе из листинга 6.1 применяется тип long, что позволяет вводить большие числа. Для согласованности переменная sum инициализируется значением 0L (ноль типа long), а не 0 (ноль типа int), хотя автоматическое преобразование типов в С позволяет указывать просто 0.

Ниже показан пример выполнения этой программы:

```
Введите целое число для последующего суммирования (или q для завершения
программы): 44
Введите следующее целое число (или q для завершения программы): 33
Введите следующее целое число (или q для завершения программы): 88
Введите следующее целое число (или q для завершения программы): 121
Введите следующее целое число (или q для завершения программы): q
Сумма введенных целых чисел равна 286.
```

Комментарии к программе

Для начала взглянем на цикл while. Условием проверки этого цикла является следующее выражение:

```
status == 1
```

В языке С символы == представляют *операцию равенства*, т.е. это выражение проверяет, равно ли значение переменной status числу 1. Не путайте его с выражением status = 1, которое присваивает переменной status значение 1. Благодаря условию проверки status == 1, цикл повторяется до тех пор, пока переменная status имеет значение 1. На каждой итерации в цикле текущее значение num добавляется к значению переменной sum, так что sum хранит промежуточную сумму. Когда переменная status получит значение, отличное от 1, цикл завершается, и программа выводит финальное значение sum.

Чтобы программа работала корректно, на каждой итерации цикла она должна получать новое значение для переменной num и переустанавливать переменную status. Это достигается за счет использования двух свойств функции scanf(). Во-первых, функция scanf() вызывается с целью чтения нового значения для num. Во-вторых, возвращаемое значение scanf() применяется для выяснения, была ли попытка чтения успешной. Как отмечалось в главе 4, функция scanf() возвращает количество успешно прочитанных элементов. Если scanf() успешно считывает целое число, она помещает его в переменную num и возвращает значение 1, которое присваивается переменной status. (Обратите внимание, что входное значение попадает в num, а не в status.) В результате обновляются значения num и status, после чего цикл while переходит на следующую итерацию. Если вы введете нечисловое значение, такое как q, то функция scanf() не обнаружит целого числа при вводе и возвратит значение 0, которое затем получит переменная status. Входной символ q из-за того, что он не является числом, возвращается во входную очередь; он вообще не считывается. (В действительности цикл завершается по причине ввода любого нечислового значения, а не только q, но предложение ввести q является более простой инструкцией для пользователя, чем сообщение о вводе любого нечислового значения.)

Если перед попыткой преобразовать значение функция scanf() сталкивается с проблемой (например, возникает конец файла или сбой оборудования), она возвращает специальное значение EOF (end of file — конец файла), которое обычно определено как -1. Это значение также приводит к прекращению цикла.

Такое двойное использование функции scanf() позволяет избежать трудноразрешимого аспекта интерактивного ввода в цикле: каким образом сообщить циклу о том, когда он должен прекратиться? Предположим, например, что scanf() не имеет

возвращаемого значения. Тогда единственное, что изменяется на каждой итерации – это значение переменной num. Значение num можно было бы задействовать при определении момента завершения цикла, скажем, указав num > 0 (num больше 0) или num != 0 (num не равно 0) в условии проверки, но это воспрепятствовало бы вводу определенных значений, таких как -3 или 0. Взамен можно было бы добавить в цикл дополнительный код, например, выдающий на каждой итерации запрос "Намерены ли вы продолжать? <да/нет>", после чего проверять, ввел ли пользователь утвердительный ответ. Однако такой подход выглядит несколько громоздким и замедляет ввод. Применение возвращаемого значения scanf() позволяет избежать этих проблем.

Теперь давайте подробнее рассмотрим структуру этой программы. Ее можно кратко описать следующим образом:

```
инициализировать переменную sum значением 0
выдать пользователю приглашение на ввод
прочитать входные данные
пока входное значение представляет собой целое число,
   добавить входное значение к значению sum,
   выдать пользователю приглашение на ввод,
   затем прочитать следующий ввод
по завершении ввода вывести значение переменной sum
```

Кстати, приведенное описание является примером *псевдокода*, который представляет собой искусство выражения программы на естественном языке, проводящее параллель с формами на языке компьютера. Псевдокод удобен при разработке логики программы. После того как логика выглядит правильной, псевдокод можно транслировать в действительный программный код. Одно из преимуществ псевдокода заключается в том, что он позволяет сосредоточиться на логике и организации программы, одновременно не беспокоясь о том, как выразить нужные идеи на языке программирования. Например, в показанном выше псевдокоде блоки указываются с помощью отступов, и совершенно не играет роли, что синтаксис C требует фигурных скобок. Еще одно достоинство псевдокода состоит в том, что он не привязан к конкретному языку программирования, благодаря чему один и тот же псевдокод можно транслировать на разные языки.

В любом случае, поскольку while является циклом с предусловием, программа должна получить входные данные и проверить значение переменной status *до того*, как будет произведен вход в тело цикла. Именно по этой причине в программе имеется вызов функции scanf() перед while. Чтобы цикл мог продолжаться, внутри него должен присутствовать оператор чтения, который позволит определить значение переменной status для следующего входного значения. В связи с этим оператор scanf() присутствует также в конце цикла while; он подготавливает цикл к следующей итерации. Приведенный ниже псевдокод можно считать стандартным форматом цикла:

```
получить первое значение, предназначенное для проверки
пока проверка проходит успешно,
   обработать значение
   получить следующее значение
```

Цикл чтения в стиле C

Программу в листинге 6.1 можно было бы написать на Pascal, BASIC или FORTRAN с тем же самым проектом, представленным с помощью псевдокода. Тем не менее, язык C предлагает сокращение. Конструкция

```
status = scanf("%ld", &num);
while (status == 1)
{
   /* действия, выполняемые в цикле */
   status = scanf("%ld", &num);
}
```

может быть заменена следующей:

```
while (scanf("%ld", &num) == 1)
{
   /* действия, выполняемые в цикле */
}
```

Во второй форме функция `scanf()` используется двумя разными способами одновременно. Во-первых, вызов функции в случае успешного завершения помещает значение в `num`. Во-вторых, возвращаемое значение этой функции (которое равно 1 или 0 и не является значением `num`) управляет циклом. Поскольку условие цикла проверяется на каждой итерации, то и функция `scanf()` вызывается на каждой итерации, предоставляя новое значение `num` и обеспечивая новую проверку. Другими словами, возможности синтаксиса С позволяют заменить стандартный формат цикла следующей компактной версией:

```
пока получение и проверка значения завершается успешно,
    обработать значение
```

А теперь рассмотрим оператор `while` более формально.

Оператор while

Общая форма цикла `while` имеет следующий вид:

```
while (выражение)
     оператор
```

Часть *оператор* может быть простым оператором, завершающимся точкой с запятой, либо составным оператором, заключенным в фигурные скобки.

До сих пор в примерах в качестве части *выражение* применялись выражения отношений, т.е. сравнивались значения. В общем случае здесь можно использовать любое выражение. Если *выражение* истинно (или в общем случае имеет ненулевое значение), то часть *оператор* выполняется один раз и затем *выражение* проверяется снова. Такой повторяющийся процесс проверки и выполнения повторяется до тех пор, пока *выражение* не станет ложным (получит нулевое значение). Каждый процесс проверки и выполнения называется *итерацией* (рис. 6.1).

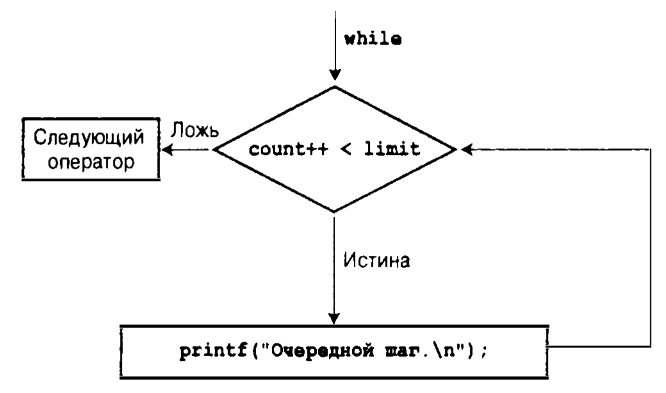


Рис. 6.1. Структура цикла `while`

Завершение цикла while

Мы подошли к *ключевому* моменту, связанному с циклами while: при построении цикла while должен быть предусмотрен код, который изменяет значение проверочного выражения, чтобы оно в конечном итоге стало ложным. В противном случае цикл никогда не закончится. (В действительности для завершения цикла можно применить операторы break и if, но об этом речь пойдет позже.) Рассмотрим следующий пример:

```
index = 1;
while (index < 5)
   printf("Доброе утро!\n");
```

Предыдущий фрагмент программы выводит это ободряющее сообщение бесконечно. Почему? Причина в том, что внутри цикла нет ничего, что изменило бы первоначальное значение index, равное 1. Теперь взгляните на такой фрагмент кода:

```
index = 1;
while (--index < 5)
   printf("Доброе утро!\n");
```

Он немногим лучше первого. Значение переменной index изменяется, но не в том направлении! Но во всяком случае эта версия кода обеспечит завершение цикла после того, как значение переменной index упадет ниже минимального отрицательного числа, поддерживаемого системой, и станет наибольшим из возможных положительным значением. (В программе toobig.c из главы 3 было продемонстрировано, что добавление 1 к максимальному положительному числу обычно дает отрицательное число, а вычитание 1 из минимального отрицательного числа, как правило, приводит к получению положительного числа.)

Когда цикл завершается?

Важно понимать, что решение прекратить или продолжить выполнение цикла принимается только после того, как вычислено условие проверки. В качестве примера рассмотрим программу, приведенную в листинге 6.2.

Листинг 6.2. Программа when.c

```
// when.c -- когда цикл завершается?
#include <stdio.h>
int main(void)
{
   int n = 5;
   while (n++ < 7);                  // строка 7
   {
      printf("n = %d\n", n);
      n++;                           // строка 10
      printf("Теперь n = %d\n", n);  // строка 11
   }
   printf("Цикл завершен.\n");
   return 0;
}
```

Запуск программы из листинга 6.2 дает следующий вывод:

```
n = 5
Теперь n = 6
n = 6
Теперь n = 7
Цикл завершен.
```

Переменная n впервые получает значение 7 в строке 10 во время второй итерации цикла. Однако цикл не прекращается немедленно. Вместо этого завершается текущая итерация (строка 11) и выход из цикла происходит только после того, как условное выражение в строке 7 будет вычислено в третий раз. (Во время первой проверки переменная n имела значение 5, а во время второй – значение 6.)

Оператор while: цикл с предусловием

Цикл while – это *условный* цикл, использующий входное условие (предусловие). Цикл называется "условным", поскольку выполнение его операторной части зависит от условия, описанного условным выражением, таким как (index < 5). Это выражение представляет собой *предусловие*, поскольку оно должно быть удовлетворено, прежде чем произойдет вход в тело цикла. В ситуациях, подобных приведенной далее, управление никогда не войдет в тело цикла, т.к. условие ложно с самого начала:

```
index = 10;
while (index++ < 5)
   printf("Удачного дня!\n");
```

Стоит изменить первую строку следующим образом:

```
index = 3;
```

и цикл начнет выполняться.

Особенности синтаксиса

Во время применения оператора while имейте в виду, что частью цикла является только один оператор, простой или составной, который следует за условием проверки. Отступы предназначены для помощи читателю, а не компьютеру. В листинге 6.3 показано, что может произойти, если забыть об этом.

Листинг 6.3. Программа while1.c

```
/* while1.c -- следите за фигурными скобками */
/* неправильное кодирование может приводить к бесконечному циклу */
#include <stdio.h>
int main(void)
{
   int n = 0;

   while (n < 3)
      printf("n равно %d\n", n);
      n++;
   printf("Это все, что делает данная программа.\n");

   return 0;
}
```

Запуск программы из листинга 6.3 порождает следующий вывод:

```
n равно 0
n равно 0
n равно 0
n равно 0
n равно 0
.
.
.
```

и т.д., пока вы не прервете ее выполнение.

Хотя в этом примере оператор n++; набран с отступом, он не заключен в фигурные скобки вместе с предыдущим оператором. Таким образом, в состав цикла входит только один оператор вывода, находящийся непосредственно после условия проверки. Переменная n никогда не изменится, условие n < 3 навсегда останется истинным, и вы получите цикл, который будет продолжать выводить сообщение до тех пор, пока вы не прервете выполнение программы. Это пример *бесконечного цикла*, из которого нельзя выйти без внешнего вмешательства.

Всегда помните, что сам по себе оператор while, даже если в нем используется составной оператор, синтаксически считается одним оператором. Он простирается от ключевого слова while до первой точки с запятой или до закрывающей фигурной скобки при наличии составного оператора.

Расставляя точки с запятой, будьте внимательны. В качестве иллюстрации взгляните на код, представленный в листинге 6.4.

Листинг 6.4. Программа while2.c

```
/* while2.c -- правильно расставляйте точки с запятой */
#include <stdio.h>
int main(void)
{
   int n = 0;
   while (n++ < 3);              /* строка 7 */
      printf("n равно %d\n", n);  /* строка 8 */
   printf("Это все, что делает данная программа.\n");
   return 0;
}
```

Программа, представленная в листинге 6.4, генерирует следующий вывод:

```
n равно 4
Это все, что делает данная программа.
```

Как упоминалось ранее, цикл заканчивается первым оператором, простым или составным, который следует непосредственно за условием проверки. Поскольку в строке 7 за условием проверки находится точка с запятой, именно в этом месте цикл и заканчивается, т.к. отдельно стоящая точка с запятой считается оператором. Оператор вывода в строке 8 не является частью цикла, поэтому значение переменной n увеличивается на каждой итерации цикла, но выводится только после выхода из цикла.

В этом примере после условия проверки следует *пустой оператор*, т.е. оператор, который ничего не делает. В языке С отдельно стоящая точка с запятой представляет пустой оператор. Временами программисты намеренно применяют оператор while с пустым оператором, поскольку вся полезная работа выполняется при проверке условия. Для примера предположим, что вы хотите пропустить входные данные до первого символа, который не является пробельным или цифровым. Можно воспользоваться таким циклом:

```
while (scanf("%d", &num) == 1)
  ;   /* пропустить целочисленный ввод */
```

До тех пор, пока функция scanf() считывает целое число, она возвращает 1, и цикл продолжается. Обратите внимание, что для ясности точка с запятой (пустой оператор) должна размещаться в следующей строке, а не вместе с оператором while. Это упрощает распознавание пустого оператора, а также напоминает о том, что он включен преднамеренно. Но еще лучше применять оператор continue, который обсуждается в следующей главе.

Сравнение: операции и выражения отношений

Поскольку циклы while часто полагаются на проверочные выражения, которые делают сравнения, эти выражения сравнения заслуживают более детального рассмотрения. Такие выражения называются *выражениями отношений*, а операции, которые в них появляются — *операциями отношений*. Мы уже пользовались несколькими такими операциями; в табл. 6.1 приведен полный список операций отношений в С. Здесь раскрыты почти все возможности в плане взаимоотношений между числами. (Взаимоотношения между числами, даже комплексными, не так сложны, как между людьми.)

Таблица 6.1. Операции отношений

Операция	Описание
<	Меньше
<=	Меньше или равно
==	Равно
>=	Больше или равно
>	Больше
!=	Не равно

Операции отношений применяются для построения выражений отношения, используемых в операторах while и в других операторах языка С, которые мы будем обсуждать позже. Эти операторы проверяют истинность или ложность значения. Ниже показаны три не связанных между собой оператора, которые содержат примеры выражений отношений. Мы надеемся, что их смысл должен быть понятен.

```
while (number < 6)
{
   printf("Число слишком мало.\n");
   scanf("%d", &number);
}
while (ch != '$')
{
   count++;
   scanf("%c", &ch);
}
while (scanf("%f", &num) == 1)
sum = sum + num;
```

Во втором примере обратите внимание на то, что в выражениях отношений могут использоваться также и символы. При сравнении задействуются машинные коды символов (предположительно ASCII). Однако операции отношений нельзя применять для сравнения строк. В главе 11 вы узнаете, что должно использоваться для сравнения строк.

Операции отношений могут также применяться с числами с плавающей запятой. Однако имейте в виду, что при сравнении чисел с плавающей вы должны использовать только операции < и >. Это объясняется тем, из-за ошибок округления два числа могут оказаться неравными, хотя логически они должны быть равны. Например, совершенно очевидно, что произведение чисел 3 и $1/3$ равно 1.0. Но если выразить число $1/3$ в виде десятичной дроби с шестью значащими цифрами, то произведением

будет .999999, что не равно в точности 1. Функция fabs(), объявленная в заголовочном файле math.h, может быть удобной при проверках, в которых участвуют числа с плавающей запятой. Эта функция возвращает абсолютное значение величины с плавающей запятой, т.е. значение без алгебраического знака. Например, с помощью кода, подобного показанному в листинге 6.5, можно проверить, насколько число близко к желаемому результату.

Листинг 6.5. Программа cmpflt.c

```
// cmpflt.c -- сравнение чисел с плавающей запятой
#include <math.h>
#include <stdio.h>
int main(void)
{
   const double ANSWER = 3.14159;
   double response;

   printf("Каково значение числа pi?\n");
   scanf("%lf", &response);
   while (fabs(response - ANSWER) > 0.0001)
   {
      printf("Введите значение еще раз.\n");
      scanf("%lf", &response);
   }
   printf("Достаточно близко!\n");

   return 0;
}
```

В цикле продолжается уточнение ответа до тех пор, пока разница между ответом и корректным значением не окажется в пределах 0.0001:

```
Каково значение числа pi?
3.14
Введите значение еще раз.
3.1416
Достаточно близко!
```

Каждое условное выражение получает оценку "истина" или "ложь" (но не "может быть"). В результате возникает интересный вопрос, о котором речь пойдет в следующем разделе.

Что такое истина?

Вы можете получить ответ на этот извечный вопрос, во всяком случае, когда дело касается языка C. Вспомните, что выражение в C всегда имеет значение. Как демонстрируется в примере, показанном в листинге 6.6, это справедливо даже для выражений отношений. В данном примере выводятся значения двух выражений отношения, одно из которых истинное, а другое — ложное.

Листинг 6.6. Программа t_and_f.c

```
/* t_and_f.c -- истинные и ложные значения в языке C */
#include <stdio.h>
int main(void)
{
   int true_val, false_val;

   true_val = (10 > 2);    // значение истинного отношения
```

```
    false_val = (10 == 2);  // значения ложного отношения
    printf("true = %d; false = %d \n", true_val, false_val);

    return 0;
}
```

В листинге 6.6 двум переменным присваиваются значения двух выражений отношений. С целью простоты переменной `true_val` присваивается значение истинного выражения, а переменной `false_val` – значение ложного выражения. Запуск этой программы дает следующий простой вывод:

```
true = 1; false = 0
```

Теперь все должно проясниться. В языке С истинное выражение имеет значение 1, а ложное выражение – 0. И действительно, в определенных программах на С для циклов, которые должны выполняться бесконечно, используется следующая конструкция, т.к. 1 всегда означает истинное значение:

```
while (1)
{
   ...
}
```

Что еще является истинным?

Если 1 или 0 допускается использовать в качестве условия проверки в операторе `while`, то можно ли применять для этих целей другие числа? И если да, то что произойдет? Давайте поэкспериментируем на примере программы, показанной в листинге 6.7.

Листинг 6.7. Программа `truth.c`

```
// truth.c -- какие значения являются истинными?
#include <stdio.h>
int main(void)
{
    int n = 3;

    while (n)
        printf("%2d является истинным\n", n--);
    printf("%2d является ложным\n", n);

    n = -3;
    while (n)
        printf("%2d является истинным\n", n++);
    printf("%2d является ложным\n", n);

    return 0;
}
```

Получаются следующие результаты:

```
 3 является истинным
 2 является истинным
 1 является истинным
 0 является ложным
-3 является истинным
-2 является истинным
-1 является истинным
 0 является ложным
```

Первый цикл выполняется, когда переменная n принимает значения 3, 2 и 1, но прекращает выполнение, когда n равна 0. Подобным же образом второй цикл выполняется, когда переменная n принимает значения -3, -2 и -1, но завершается, как только n становится равной 0. В общем случае *все* ненулевые значения рассматриваются как истинные, а ложным считается только 0. В языке С истина имеет очень широкое толкование!

Говоря по-другому, цикл while выполняется до тех пор, пока в результате вычисления его условия проверки получается ненулевое значение. Это обстоятельство перемещает условия проверки из рамок "истина/ложь" в числовую область. Имейте в виду, что условное выражение принимает значение 1, если оно истинно, и 0, если ложно, поэтому все выражения подобного рода в действительности являются числовыми.

Многие программирующие на С пользуются этим свойством условий проверки. Например, конструкцию while (goats != 0) можно заменить while (goats), поскольку выражения (goats != 0) и (goats) принимают значение 0, т.е. ложное, только когда переменная goats равна 0. Вы должны в достаточной мере попрактиковаться с формой while (goats), чтобы она стала привычной.

Затруднения с понятием истины

Довольно широкое толкование понятия истины в С может приводить к затруднениям. Например, давайте внесем одно тонкое изменение в программу из листинга 6.1, получив в результате программу, показанную в листинге 6.8.

Листинг 6.8. Программа trouble.c

```
// trouble.c -- неправильное применение операции =
// приводит к возникновению бесконечного цикла
#include <stdio.h>
int main(void)
{
    long num;
    long sum = 0L;
    int status;
    printf("Введите целое число для последующего суммирования ");
    printf("(или q для завершения программы): ");
    status = scanf("%ld", &num);
    while (status = 1)
    {
        sum = sum + num;
        printf("Введите следующее целое число (или q для завершения программы): ");
        status = scanf("%ld", &num);
    }
    printf("Сумма введенных целых чисел равна %ld.\n", sum);
    return 0;
}
```

Запуск программы из листинга 6.8 дает следующий вывод:

```
Введите целое число для последующего суммирования (или q для завершения программы): 20
Введите следующее целое число (или q для завершения программы): 5
Введите следующее целое число (или q для завершения программы): 30
Введите следующее целое число (или q для завершения программы): q
Введите следующее целое число (или q для завершения программы):
Введите следующее целое число (или q для завершения программы):
Введите следующее целое число (или q для завершения программы):
Введите следующее целое число (или q для завершения программы):
.
.
.
```

и т.д. до тех пор, пока вы не прервете выполнение программы.

К такому плачевному результату привело изменение, внесенное в условие проверки оператора while, когда выражение status == 1 было заменено выражением status = 1. Второе выражение — это оператор присваивания, который устанавливает переменную status в 1. Более того, значением оператора присваивания является значение его левой части, поэтому status = 1 имеет то же самое числовое значение 1. Следовательно, с практической точки зрения этот цикл while дает такой же результат, как и while (1) — т.е. цикл никогда не завершится. В случае ввода q переменная status устанавливается в 0, однако при проверке условия цикла status получает значение 1 и инициирует следующую итерацию.

Может возникнуть вопрос, по какой причине, учитывая, что программа выполняется в цикле, пользователь лишен возможности ввести какие-то дополнительные данные после ввода символа q. Когда функция scanf() не может прочитать данные в указанной форме, она оставляет этот не соответствующий ее требованиям ввод на месте для его чтения в следующий раз. Следовательно, после неудавшейся попытки чтения символа q как целого числа функция scanf() оставляет q на месте. На следующей итерации цикла scanf() пытается выполнить считывание с того места, где оно было остановлено в последний раз — там, где остался символ q. Функция scanf() снова не может прочитать q как целое число, так что этот пример демонстрирует не только возникновение бесконечного цикла, но также и цикла безуспешных попыток чтения. Словом, результат получается довольно плачевный, и хорошо, что компьютеры пока еще лишены чувств. Слепое следование неразумным инструкциям для компьютера столь же бесперспективно, как и попытка предсказать ситуацию на фондовой бирже на ближайшее десятилетие.

Не используйте знак = вместо ==. В некоторых языках программирования (скажем, BASIC) для представления операции присваивания и операции проверки на равенство применяется один и тот же символ, однако это совершенно разные операции (рис. 6.2).

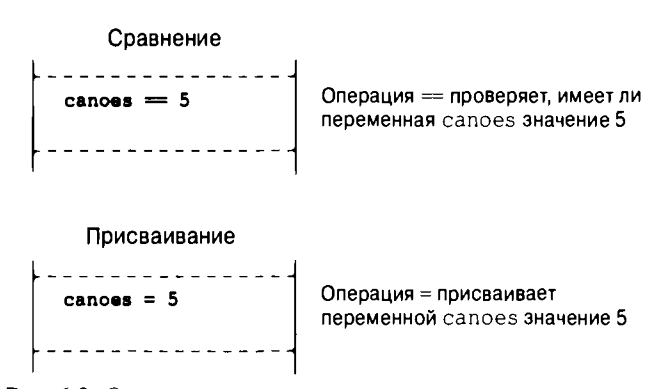


Рис. 6.2. Операция отношения == и операция присваивания =

Операция присваивания устанавливает значение переменной, указанной слева от знака операции. С другой стороны, операция проверки на равенство выясняет, эквивалентны ли левая и правая части операции. Она не изменяет значение переменной, расположенной слева, если она указана. Рассмотрим пример:

```
canoes = 5      ← Присваивает значение 5 переменной canoes
canoes == 5     ← Проверяет, имеет ли переменная canoes значение 5
```

Будьте внимательны во время применения операции. Компилятор позволит вам использовать неподходящую форму, выдавая результат, который будет отличаться от ожидаемого. (Однако из-за того, что очень много программистов слишком часто

неправильно применяли операцию =, большинство компиляторов выводит на экран предупреждение о том, что возможно предполагалась другая операция.) Если одно из сравниваемых значений является константой, его можно поместить слева от знака операции сравнения, чтобы облегчить выявление ошибки:

```
5 = canoes      ← Синтаксическая ошибка
5 == canoes     ← Проверяет, имеет ли переменная canoes значение 5
```

Смысл в том, что присваивать значение константе не допускается, поэтому компилятор трактует такую операцию присваивания как синтаксическую ошибку. Многие практикующие программисты в выражениях проверки на равенство первой указывают константу.

Итак, операции отношения используются для построения условных выражений. Выражения отношений имеют значение 1, если они истинны, и 0, если ложны. В операторах (таких как while и if), где обычно применяются выражения отношений в качестве условий проверки, могут использоваться любые выражения, при этом их ненулевые значения интерпретируются как "истина", а нулевые – как "ложь".

Новый тип _Bool

Переменные, предназначенные для представления истинных и ложных значений, в языке C традиционно имеют тип int. В стандарте C99 для переменных такого вида был введен тип _Bool. Тип получил свое название в честь Джорджа Буля (George Boole), английского математика, который разработал алгебраическую систему, позволяющую формулировать и решать логические задачи. В программировании переменные, представляющие истинные и ложные значения, известны как *булевские*, так что именем типа для этих переменных в языке C является _Bool. Переменная типа _Bool может иметь только значения 1 ("истина") и 0 ("ложь"). Если вы попытаетесь присвоить переменной _Bool ненулевое числовое значение, то она получит значение 1, отражая тот факт, что любое ненулевое значение в C трактуется как "истина".

В листинге 6.9 исправлено условие проверки, указанное в листинге 6.8, и переменная status типа int заменена переменной input_is_good типа _Bool. Назначение булевским переменным имен, по которым ясно, что они принимают истинные и ложные значения, является общей практикой.

Листинг 6.9. Программа boolean.c

```
// boolean.c -- использование переменной типа _Bool
#include <stdio.h>
int main(void)
{
   long num;
   long sum = 0L;
   _Bool input_is_good;
   printf("Введите целое число для последующего суммирования ");
   printf("(или q для завершения программы): ");
   input_is_good = (scanf("%ld", &num) == 1);
   while (input_is_good)
   {
      sum = sum + num;
      printf("Введите следующее целое число (или q для завершения программы): ");
      input_is_good = (scanf("%ld", &num) == 1);
   }
   printf("Сумма введенных целых чисел равна %ld.\n", sum);
   return 0;
}
```

Обратите внимание на то, как в коде присваивает результат сравнения переменной:

```
input_is_good = (scanf("%ld", &num) == 1);
```

Такое присваивание имеет смысл, поскольку операция == возвращает значение 1 или 0. Кстати, круглые скобки, заключающие в себе выражение ==, не нужны, поскольку операция == имеет более высокий приоритет, чем =; тем не менее, они способствуют улучшению читабельности кода. Кроме того, взгляните, насколько изменение имени переменной делает проверку цикла while более понятной:

```
while (input_is_good)
```

Стандарт C99 также предлагает заголовочный файл stdbool.h, в котором bool сделан псевдонимом типа _Bool и определены символические константы true и false для значений 1 и 0. Включение этого заголовочного файла позволяет писать код, совместимый с программами на языке C++, где bool, true и false являются ключевыми словами.

Если ваша система не поддерживает тип _Bool, его можно заменить типом int, и приведенный выше пример будет работать так же.

Приоритеты операций отношений

Приоритет операций отношений ниже приоритета арифметических операций, + и -, но выше, чем у операции присваивания. Это значит, например, что следующее выражение:

```
x > y + 2
```

эквивалентно

```
x > (y + 2)
```

И также значит, что

```
x = y > 2
```

эквивалентно

```
x = (y > 2)
```

Другими словами, переменной x присваивается значение 1, если y больше 2, и 0 — в противном случае; переменной x не присваивается значение y.

Операции отношений имеют больший приоритет, чем операции присваивания. Поэтому

```
x_bigger = x > y;
```

эквивалентно

```
x_bigger = (x > y);
```

Операции отношений по своему приоритету делятся на две группы.

Группа с большим приоритетом: < <= > >=

Группа с меньшим приоритетом: == !=

Подобно большинству других операций, операции отношений выполняются слева направо. Поэтому

```
ex != wye == zee
```

эквивалентно

```
(ex != wye) == zee
```

Сначала осуществляется проверка на неравенство значений переменных ex и wye. Затем полученное значение, которое может быть равно 1 или 0 ("истина" или "ложь"), сравнивается со значением zee. Мы вовсе не предлагаем вам применять конструкции подобного рода, но считаем своим долгом дать соответствующее пояснение.

В табл. 6.2 показаны приоритеты операций, представленных до сих пор. В справочном разделе II приложения Б приведена информация по приоритетам всех операций.

Таблица 6.2. Приоритет операций

Операции (в порядке убывания приоритета)	Ассоциативность
`( )`	Слева направо
`- + ++ -- sizeof` (*тип*) (все унарные)	Справа налево
`* / %`	Слева направо
`+ -`	Слева направо
`< > <= >=`	Слева направо
`== !=`	Слева направо
`=`	Справа налево

Сводка: оператор `while`

Ключевое слово

`while`

Общий комментарий

Оператор `while` создает цикл, который повторяется до тех пор, пока проверочное выражение не станет ложным, или нулевым. Оператор `while` представляет собой цикл с предусловием; это значит, что решение относительно очередной итерации цикла принимается перед проходом. Следовательно, вполне возможно, что цикл вообще не будет выполнен. Операторная часть цикла может быть простым или составным оператором.

Форма записи

```
while (выражение)
   оператор
```

Часть *оператор* повторяется до тех пор, пока *выражение* не станет ложным или равным 0.

Примеры

```
while (n++ < 100)
   printf(" %d %d\n",n, 2 * n + 1); // одиночный оператор

while (fargo < 1000)
{                                    // составной оператор
   fargo = fargo + step;
   step = 2 * step;
}
```

Сводка: операции отношений и условные выражения

Операции отношений

Каждая операция отношения сравнивает значение в ее левой части со значением в ее правой части:

<	меньше
<=	меньше или равно
==	равно
>=	больше или равно
>	больше
!=	не равно

Условные выражения

Простое условное выражение состоит из знака операции отношения и операндов слева и справа. Если выражение истинно, то условное выражение имеет значение `1`. Если отношение ложно, то условнot выражения получает значение `0`.

Примеры

`5 > 2` истинно и принимает значение `1`.

`(2 + a) == a` ложно и принимает значение `0`.

Неопределенные циклы и циклы со счетчиком

Некоторые примеры цикла `while` представляли собой *неопределенные циклы*. Это означает, что заранее нельзя сказать, сколько раз цикл выполнится до того, как выражение станет ложным. Например, когда в листинге 6.1 использовался интерактивный цикл для суммирования целых чисел, заранее не было известно, сколько чисел будет введено. Тем не менее, в других примерах применялись *циклы со счетчиком*. Такие циклы выполняют заранее известное количество итераций. В листинге 6.10 приведен пример оператора цикла `while` со счетчиком.

Листинг 6.10. Программа `sweetie1.c`

```
// sweetie1.c -- цикл со счетчиком
#include <stdio.h>
int main(void)
{
   const int NUMBER = 22;
   int count = 1;                     // инициализация
   while (count <= NUMBER)            // проверка
   {
      printf("Будь моим другом!\n");  // действие
      count++;                        // обновление значения count
   }
   return 0;
}
```

И хотя форма цикла, использованная в листинге 6.10, работает прекрасно, это не самый лучший выбор в подобной ситуации, поскольку действия, определяющие цикл, не собраны вместе. Давайте исследуем этот вопрос более глубоко.

Чтобы организовать цикл, который должен быть повторен фиксированное количество раз, необходимо выполнить три действия.

1. Инициализировать счетчик.
2. Сравнить показание счетчика с некоторой граничной величиной.
3. Инкрементировать значение счетчика на каждом проходе цикла.

О сравнении позаботится условие цикла while. Операция инкремента отвечает за увеличение значения счетчика. В листинге 6.10 инкрементирование делается в конце цикла. Такой подход устраняет возможность случайно пропустить действие инкрементирования. Следовательно, было бы лучше объединить действия по проверке и обновлению в одно выражение, применив конструкцию count++ <= NUMBER, но инициализация счетчика по-прежнему выполняется за пределами цикла, сохраняя вероятность забыть сделать это. Опыт нас учит: то, что может случиться, рано или поздно *произойдет*, поэтому давайте более подробно рассмотрим управляющий оператор, который позволяет избежать таких проблем.

Цикл for

Цикл for собирает все три указанных выше действия (инициализацию, проверку и обновление) в одном месте. Используя цикл for, предыдущую программу можно заменить кодом, приведенным в листинге 6.11.

Листинг 6.11. Программа sweetie2.c

```
// sweetie2.c -- цикл for со счетчиком
#include <stdio.h>
int main(void)
{
   const int NUMBER = 22;
   int count;

   for (count = 1; count <= 100; count++)
      printf("Будь моим другом!\n");

   return 0;
}
```

В круглых скобках, следующих за ключевым словом for, содержатся три выражения, разделенные двумя точками с запятой. Первое выражение — это инициализация. Она осуществляется только один раз при первом запуске цикла for. Второе выражение представляет собой условие проверки; оно вычисляется перед каждым потенциальным проходом цикла. Когда выражение имеет значение false (когда значение счетчика count больше, чем NUMBER), цикл завершается. Третье выражение, которое выполняет изменение или обновление, вычисляется в конце каждой итерации. В листинге 6.10 оно применяется для инкрементирования значения count, но этим его использование не ограничивается. Оператор for завершается следующим за ним простым или составным оператором. Каждое из трех выражений является полным, так что любой побочный эффект в управляющем выражении, такой как инкремент значения переменной, происходит до вычисления другого выражения. Структура цикла for иллюстрируется на рис. 6.3.

В качестве еще одного примера в листинге 6.12 показана программа, в которой цикл for применяется для вывода таблицы кубов.

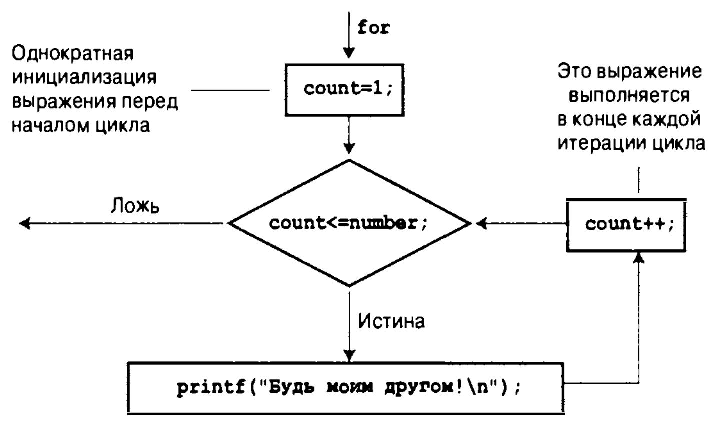


Рис. 6.3. Структура цикла `for`

Листинг 6.12. Программа `for_cube.c`

```
/* for_cube.c -- использование цикла for для построения таблицы кубов */
#include <stdio.h>
int main(void)
{
   int num;
   printf("    n   n в кубе\n");
   for (num = 1; num <= 6; num++)
      printf("%5d %5d\n", num, num*num*num);
   return 0;
}
```

Программа в листинге 6.12 выводит целые числа от 1 до 6 и результат их возведения в куб.

```
n     n в кубе
1       1
2       8
3      27
4      64
5     125
6     216
```

Первая строка цикла `for` немедленно сообщает всю информацию о параметрах цикла: начальное значение `num`, конечное значение `num` и величину, на которую `num` увеличивается при каждом проходе цикла.

Использование цикла `for` для повышения гибкости

Хотя цикл `for` выглядит похожим на цикл `DO` в языке FORTRAN, цикл `FOR` в языке Pascal и цикл `FOR...NEXT` в языке BASIC, он обладает гораздо большей гибкостью, чем любой из них. Эта гибкость является результатом того, как могут применяться три выражения в спецификации оператора `for`. В примерах, приведенных до сих пор, первое выражение использовалось для инициализации счетчика, второе выражение — для установки предельного значения счетчика и третье выражение — для увеличения значения счетчика на 1. Применяемый подобным образом оператор `for` языка C во многом подобен упомянутым выше операторам. Тем не менее, он обладает и множеством других возможностей, девять из которых описаны далее.

- Вы можете применять операцию декремента для реализации обратного отсчета значений счетчика вместо прямого.

```
/* for_down.c */
#include <stdio.h>
int main(void)
{
   int secs;
   for (secs = 5; secs > 0; secs--)
   printf("%d секунд(ы)!\n", secs);
      printf("Ключ на зажигание!\n");
   return 0;
}
```

Ниже показан вывод:

```
5 секунд(ы)!
4 секунд(ы)!
3 секунд(ы)!
2 секунд(ы)!
1 секунд(ы)!
Ключ на зажигание!
```

- При желании можно считать двойками, десятками и т.д.

```
/* for_13s.c */
#include <stdio.h>
int main(void)
{
   int n;       // счет с интервалом 13, начиная с 2
   for (n = 2; n < 60; n = n + 13)
      printf("%d \n", n);
   return 0;
}
```

Значение переменной n на каждой итерации увеличивается на 13, давая следующий вывод:

```
2
15
28
41
54
```

- Можно делать подсчет по символам, а не числам.

```
/* for_char.c */
#include <stdio.h>
int main(void)
{
   char ch;
   for (ch = 'a'; ch <= 'z'; ch++)
      printf("Значение ASCII для %c равно %d.\n", ch, ch);
   return 0;
}
```

В программе предполагается, что для символов в системе используется кодировка ASCII. Ниже показан фрагмент выходных данных:

```
Значение ASCII для a равно 97.
Значение ASCII для b равно 98.
...
```

```
Значение ASCII для x равно 120.
Значение ASCII для y равно 121.
Значение ASCII для z равно 122.
```

Программа работает, т.к. символы хранятся в памяти в виде целых чисел, поэтому цикл в любом случае имеет дело с целыми числами.

- Можно выполнять проверку условия, отличного от количества итераций. В программе `for_cube.c` строку

  ```
  for (num = 1; num <= 6; num++)
  ```

  можно заменить такой строкой:

  ```
  for (num = 1; num*num*num <= 216; num++)
  ```

  Такое условие проверки можно было бы применять, если вас больше заботит ограничение величины куба, а не количества итераций.

- Можно позволить некоторой величине расти не в арифметической, а в геометрической прогрессии; т.е. вместо добавления каждый раз фиксированного значения можно умножать на фиксированное значение.

  ```
  /* for_geo.c */
  #include <stdio.h>
  int main(void)
  {
     double debt;
     for (debt = 100.0; debt < 150.0; debt = debt * 1.1)
        printf("Теперь ваша задолженность составляет $%.2f.\n", debt);
     return 0;
  }
  ```

  В этом фрагменте кода значение переменной `debt` на каждом проходе цикла умножается на 1.1, что увеличивает его на 10%. Вывод программы имеет следующий вид:

  ```
  Теперь ваша задолженность составляет $100.00.
  Теперь ваша задолженность составляет $110.00.
  Теперь ваша задолженность составляет $121.00.
  Теперь ваша задолженность составляет $133.10.
  Теперь ваша задолженность составляет $146.41.
  ```

- В качестве третьего выражения можно использовать любое допустимое выражение. Что бы вы здесь не поместили, это будет обновляться на каждой итерации.

  ```
  /* for_wild.c */
  #include <stdio.h>
  int main(void)
  {
     int x;
     int y = 55;
     for (x = 1; y <= 75; y = (++x * 5) + 50)
        printf("%10d %10d\n", x, y);
     return 0;
  }
  ```

  Этот цикл выводит значения переменной `x` и алгебраического выражения `++x * 5 + 50`. Ниже показан вывод программы:

  ```
  1         55
  2         60
  3         65
  4         70
  5         75
  ```

Обратите внимание, что в проверке участвует переменная y, а не x. В каждом из трех выражений, управляющих циклом for, могут применяться разные переменные. (Следует отметить, что хотя этот пример является допустимым, он не соответствует хорошему стилю программирования. Программа могла бы стать более ясной, если не смешивать процесс обновления с алгебраическими вычислениями.)

- Можно даже оставить одно или несколько выражений пустыми (но не опускайте точки с запятой). Обеспечьте наличие в цикле оператора, который в конечном итоге приведет к завершению цикла.

```
/* for_none.c */
#include <stdio.h>
int main(void)
{
    int ans, n;
    ans = 2;
    for (n = 3; ans <= 25; )
        ans = ans * n;
    printf("n = %d; ans = %d.\n", n, ans);
    return 0;
}
```

Вот вывод этой программы:

```
n = 3; ans = 54.
```

Значение переменной n в цикле остается равным 3. Переменная ans начинается со значения 2, затем увеличивается до 6 и 18, а в конечном итоге получает значение 54. (Значение 18 меньше, чем 25, так что цикл for выполняет на одну итерацию больше, умножая 18 на 3 для получения 54.) Кстати, пустое управляющее выражение, находящееся посередине, считается истинным, поэтому приведенный ниже цикл выполняется бесконечно:

```
for ( ; ; )
   printf("Требуется определенное действие.\n");
```

- Первое выражение не обязательно должно инициализировать переменную. Вместо этого им могла бы быть какая-то разновидность оператора printf(). Просто запомните, что первое выражение вычисляется или выполняется только один раз, до того как начнут выполняться другие части цикла.

```
/* for_show.c */
#include <stdio.h>
int main(void)
{
    int num = 0;
    for (printf("Продолжайте вводить числа!\n"); num != 6; )
        scanf("%d", &num);
    printf("Вот то число, которое было нужно!\n");
    return 0;
}
```

Этот фрагмент кода один раз выводит первое сообщение, а затем продолжает принимать числа до тех пор, пока вы не введете 6:

```
Продолжайте вводить числа!
3
5
8
6
Вот то число, которое было нужно!
```

- Параметры выражений цикла могут изменяться с помощью действий внутри тела цикла. Например, предположим, что цикл определен следующим образом:

  ```
  for (n = 1; n < 10000; n = n + delta)
  ```

  Если после нескольких итераций программа решит, что значение `delta` слишком мало или слишком велико, то посредством оператора `if` (глава 7) внутри цикла величину `delta` можно изменить. В интерактивной программе значение `delta` может быть изменено пользователем в процессе выполнения цикла. С таким видом настройки связана и определенная опасность; к примеру, установка `delta` в 0 приведет к бесконечному циклу.

Короче говоря, свобода при выборе выражений, управляющих циклом `for`, предоставляет этому циклу способность делать намного больше, чем просто выполнение фиксированного количества операций. Полезность цикла `for` может быть еще больше увеличена посредством операций, которые мы вскоре обсудим.

Сводка: оператор `for`

Ключевое слово

`for`

Общий комментарий

В операторе `for` используются три управляющих выражения, разделяемые точками с запятой. Выражение *инициализация* вычисляется однократно до выполнения любых операторов внутри цикла. Затем вычисляется выражение *проверка*, и если оно истинно (или не равно нулю), то тело цикла выполняется один раз. Далее вычисляется выражение *обновление*, после чего снова вычисляется выражение *проверка*. Оператор `for` представляет собой цикл с предусловием — решение о проходе цикла еще раз принимается перед входом в него. Таким образом, вполне возможно, что цикл не выполнится ни разу. Часть *оператор* может быть простым или составным оператором.

Форма записи

```
for (инициализация; проверка; обновление)
   оператор
```

Цикл повторяется до тех пор, пока выражение *проверка* не станет ложным или нулевым.

Пример

```
for (n = 0;  n < 10 ; n++)
   printf(" %d %d\n", n, 2 * n + 1);
```

Дополнительные операции присваивания: +=, -=, *=, /=, %=

В языке C доступно несколько операций присваивания. Наиболее базовой из них, конечно же, является операция =, которая просто присваивает значение выражения, находящегося справа от знака операции, переменной, указанной слева от этого знака. Другие операции присваивания модифицируют переменные. В каждой такой операции имя переменной располагается слева от знака операции, а выражение — справа. Переменной присваивается новое значение, которое образовано путем корректировки ее старого значения на величину выражения, стоящего справа. Действительная корректировка зависит от операции.

Например:

scores += 20 — то же, что и scores = scores + 20

dimes -= 2 — то же, что и dimes = dimes - 2

bunnies *= 2 — то же, что и bunnies = bunnies * 2

time /= 2.73 — то же, что и time = time / 2.73

reduce %= 3 — то же, что и reduce = reduce % 3

В приведенных выше примерах применялись простые числа, но эти операции также работают и с более сложными выражениями:

x *= 3 * y + 12 — то же, что и x = x * (3 * y + 12)

Только что рассмотренные операции присваивания имеют такой же низкий приоритет, как и операция =, и этот приоритет меньше приоритета операции + или *. Низкий приоритет отражен в последнем примере, в котором 12 суммируется с 3 * y и только затем результат умножается на x.

Использовать все эти формы операции присваивания совершенно не обязательно. С другой стороны, они компактны и могут генерировать более эффективный машинный код, чем длинная форма операции. Комбинированные операции присваивания особенно полезны в ситуации, когда необходимо поместить в спецификацию цикла for сложное выражение.

Операция запятой

Операция запятой повышает гибкость цикла for, позволяя включить в его спецификацию более одного выражения инициализации или обновления. В листинге 6.13 представлена программа, которая выводит тарифы почтового обслуживания первого класса. (В 2013 году тарифы составляли 46 центов за первую унцию и 20 центов за каждую последующую унцию пересылаемого груза.)

Листинг 6.13. Программа postage.c

```
// postage.c -- тарифы почтового обслуживания первого класса
#include <stdio.h>
int main(void)
{
    const int FIRST_OZ = 46; // тариф 2013 года
    const int NEXT_OZ = 20;  // тариф 2013 года
    int ounces, cost;

    printf(" унции  тариф\n");
    for (ounces=1, cost=FIRST_OZ; ounces <= 16; ounces++,
         cost += NEXT_OZ)
       printf("%5d $%4.2f\n", ounces, cost/100.0);

    return 0;
}
```

Первые пять строк вывода программы выглядят так:

```
 унции  тариф
    1  $0.46
    2  $0.66
    3  $0.86
    4  $1.06
```

Операция запятой в программе применяется в выражениях инициализации и обновления. Ее наличие в первом выражении приводит к инициализации переменных ounces и cost. Второе ее вхождение вызывает увеличение на 1 переменной ounces и увеличение на 20 (значение константы NEXT_OZ) переменной cost на каждой итерации. Все вычисления делаются в спецификациях цикла for (рис. 6.4).

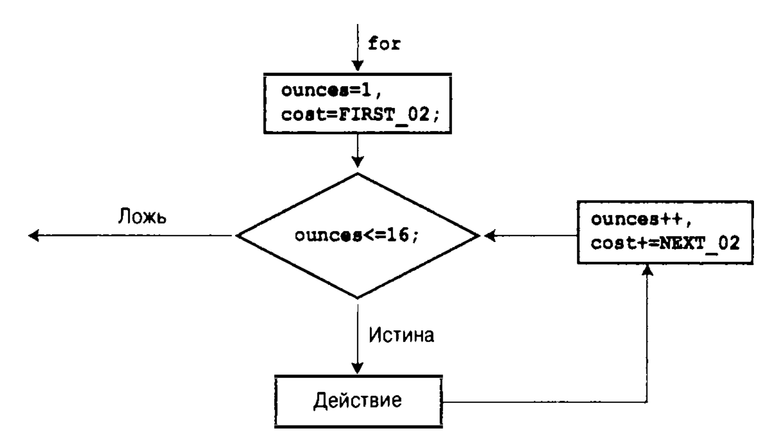


Рис. 6.4. *Операция запятой и цикл* for

Использование операции запятой не ограничивается циклами for, но именно здесь она применяется чаще всего. Эта операция обладает еще двумя свойствами. Во-первых, она гарантирует, что разделяемые ею выражения вычисляются в порядке слева направо. (Другими словами, запятая является точкой следования, поэтому все побочные эффекты слева от запятой учитываются до того, как происходит переход вправо от запятой.) Таким образом, переменная ounces инициализируется раньше переменной cost. В рассматриваемом примере порядок не имеет значения, однако он важен в ситуации, когда переменная ounces задействована в выражении для cost. Для примера взгляните на следующее выражение:

```
ounces++, cost = ounces * FIRST_OZ
```

Это приводит к инкрементированию переменной ounces и использованию нового значения ounces во втором подвыражении. Операция запятой, будучи точкой следования, гарантирует, что побочные эффекты левого подвыражения проявятся перед тем, как будет вычислено правое подвыражение.

Во-вторых, значение всего выражения, содержащего операцию запятой, является значением операнда в правой части. Результат выполнения оператора

```
x = (y = 3, (z = ++y + 2) + 5);
```

выглядит так: переменной y присваивается 3, значение y инкрементируется до 4, к этому значению 4 добавляется 2, результирующее значение 6 присваивается переменной z, к z добавляется 5 и, наконец, финальное значение переменной z, равное 11, присваивается переменной x. Объяснение, почему кто-то мог все это делать, выходит за рамки настоящей книги. С другой стороны, предположим, что вы по неосторожности указали запятую при записи числа:

```
houseprice = 249,500;
```

Здесь нет синтаксической ошибки. Взамен компилятор С интерпретирует это как выражение запятой, с houseprice = 249 в качестве левого подвыражения и 500 — в ка-

честве правого. Следовательно, значение всего выражения запятой – это выражение в правой части, а подоператор в левой части присваивает переменной houseprice значение 249. Таким образом, результат совпадает с результатом выполнения следующего кода:

```
houseprice = 249;
500;
```

Вспомните, что любое выражение становится оператором, если в его конце добавить точку с запятой, поэтому 500; является оператором, который ничего не делает.

С другой стороны, оператор

```
houseprice = (249,500);
```

присваивает переменной houseprice значение 500.

Запятая применяется также в качестве разделителя, так что запятые в выражении

```
char ch, date;
```

и в операторе

```
printf("%d %d\n", chimps, chumps);
```

представляют собой разделители, а не операции запятой.

Сводка: новые операции

Операции присваивания

Каждая из приведенных ниже операций обновляет переменную, стоящую слева от знака операции, значением, стоящим справа от этого знака.

- `+=` Добавляет величину справа от знака операции к значению слева от знака.
- `-=` Вычитает величину справа от знака операции из значения слева от знака.
- `*=` Умножает значение переменной слева от знака операции на величину справа от знака.
- `/=` Делит значение переменной слева от знака операции на величину справа от знака.
- `%=` Возвращает остаток от деления значения переменной слева от знака операции на величину справа от знака.

Пример

```
rabbits *= 1.6;
```

эквивалентна

```
rabbits = rabbits * 1.6;
```

Эти комбинированные операции присваивания имеют такой же низкий приоритет, как и традиционная операция присваивания, который меньше приоритета арифметических операций. Таким образом, следующий оператор

```
contents *= old_rate + 1.2;
```

дает тот же результат, что и такой оператор:

```
contents = contents * (old_rate + 1.2);
```

Операция запятой

Операция запятой связывает два выражения в одно и гарантирует, что выражение, находящееся слева от знака операции, вычисляется первым. Обычно она используется для того, чтобы включить как можно больше информации в управляющее выражение цикла for. Значением всего выражения является значение выражения справа от знака операции.

Пример

```
for (step = 2, fargo = 0; fargo < 1000; step *= 2)
     fargo += step;
```

Греческий философ Зенон и цикл for

Давайте посмотрим, как с помощью цикла for и операции запятой можно разрешить древний парадокс. Греческий философ Зенон утверждал, что стрела никогда не поразит своей цели. Сначала, говорил он, стрела пролетает половину пути до цели. Затем она пролетает половину оставшегося пути, затем – половину того пути, который остался, и так до бесконечности. Поскольку весь путь стрелы разбит на бесконечное количество частей, утверждал Зенон, стреле потребуется бесконечно большой промежуток времени для достижения конца пути. Однако мы сомневаемся в том, что Зенон добровольно согласился бы стать живой мишенью, чтобы доказать свою правоту.

Применим количественный подход и предположим, что стреле требуется одна секунда, чтобы пролететь первую половину пути. Затем ей понадобится 1/2 секунды, чтобы пролететь половину оставшегося пути, еще 1/4 секунды, чтобы преодолеть половину пути, который остался после этого, и т.д. Полное время пролета стрелы можно представить в виде следующей бесконечной последовательности:

```
1 + 1/2 + 1/4 + 1/8 + 1/16 + ...
```

Короткая программа в листинге 6.14 вычисляет сумму нескольких первых элементов этой последовательности. Переменная power_of_two принимает значения 1.0, 2.0, 4.0, 8.0 и т.д.

Листинг 6.14. Программа zeno.c

```
/* zeno.c -- сумма последовательности */
#include <stdio.h>
int main(void)
{
    int t_ct;       // счетчик элементов
    double time, power_of_2;
    int limit;

    printf("Введите желаемое количество элементов последовательности: ");
    scanf("%d", &limit);
    for (time=0, power_of_2=1, t_ct=1; t_ct <= limit;
                                  t_ct++, power_of_2 *= 2.0)
    {
        time += 1.0/power_of_2;
        printf("время = %f, когда количество элементов = %d.\n", time, t_ct);
    }

    return 0;
}
```

Результат выполнения программы, в ходе которого суммируются первые 15 элементов последовательности, выглядит следующим образом:

```
Введите желаемое количество элементов последовательности: 15
Время = 1.000000, когда количество элементов = 1.
Время = 1.500000, когда количество элементов = 2.
Время = 1.750000, когда количество элементов = 3.
Время = 1.875000, когда количество элементов = 4.
Время = 1.937500, когда количество элементов = 5.
Время = 1.968750, когда количество элементов = 6.
Время = 1.984375, когда количество элементов = 7.
Время = 1.992188, когда количество элементов = 8.
Время = 1.996094, когда количество элементов = 9.
```

```
Время = 1.998047, когда количество элементов = 10.
Время = 1.999023, когда количество элементов = 11.
Время = 1.999512, когда количество элементов = 12.
Время = 1.999756, когда количество элементов = 13.
Время = 1.999878, когда количество элементов = 14.
Время = 1.999939, когда количество элементов = 15.
```

Легко заметить, что хотя мы и добавляем все новые элементы, общая сумма, по-видимому, не превысит некоторой величины. И в самом деле, математики доказали, что сумма этой последовательности стремится к 2.0 по мере того, как количество просуммированных элементов стремится к бесконечности, на что указывают результаты выполнения программы. Ознакомьтесь со следующими математическими выкладками. Предположим, что S представляет собой такую сумму:

```
S = 1 + 1/2 + 1/4 + 1/8 + ...
```

Здесь многоточие означает "и т.д.". Разделив S на 2, получаем:

```
S/2 = 1/2 + 1/4 + 1/8 + 1/16 + ...
```

Вычитание второго выражения из первого дает:

```
S - S/2 = 1 +1/2 -1/2 + 1/4 -1/4 + ...
```

За исключением начального значения 1 все остальные значения образуют пары, в которых одно значение положительное, а второе – отрицательное, так что эти элементы уничтожают друг друга, в результате оставляя:

```
S/2 = 1
```

И, наконец, умножение обеих сторон на 2 дает:

```
S = 2
```

Мораль, которую можно извлечь отсюда, такова: прежде чем начинать сложные вычисления, проверьте, не нашли ли математики более простого способа делать это.

Что можно сказать о самой программе? Она показывает, что в выражении можно использовать более одной операции запятой. Вы инициализировали переменные time, power_of_2 и count. После того, как вы определили условия для цикла, программа оказалась совсем короткой.

Цикл с постусловием: do while

Циклы while и for являются циклами с предусловием. Условия проверки вычисляются *перед* каждой итерацией цикла, поэтому вполне возможно, что операторы, помещенные в цикл, никогда не выполнятся. В языке С имеется также цикл с *постусловием*, в котором проверка условия производится после прохода каждой итерации цикла, благодаря чему гарантируется выполнение операторной части цикла минимум один раз. Эта разновидность цикла называется циклом do while. В листинге 6.15 приведен пример.

Листинг 6.15. Программа do_while.c

```
/* do_while.c -- цикл с постусловием */
#include <stdio.h>
int main(void)
{
   const int secret_code = 13;
   int code_entered;
```

```
    do
    {
        printf("Чтобы войти в клуб лечения трискадекафобии,\n");
        printf("пожалуйста, введите секретный код: ");
        scanf("%d", &code_entered);
    } while (code_entered != secret_code);
    printf("Поздравляем! Вас вылечили!\n");

    return 0;
}
```

Программа в листинге 6.15 читает входные значения до тех пор, пока пользователь не введет 13. Ниже показан результат выполнения этой программы:

```
Чтобы войти в клуб лечения трискадекафобии,
пожалуйста, введите секретный код: 12
Чтобы войти в клуб лечения трискадекафобии,
пожалуйста, введите секретный код: 14
Чтобы войти в клуб лечения трискадекафобии,
пожалуйста, введите секретный код: 13
Поздравляем! Вас вылечили!
```

Эквивалентная программа, в которой применяется цикл while, была бы несколько длиннее, как можно видеть в листинге 6.16.

Листинг 6.16. Программа entry.c

```
/* entry.c -- цикл с предусловием */
#include <stdio.h>
int main(void)
{
    const int secret_code = 13;
    int code_entered;

    printf("Чтобы войти в клуб лечения трискадекафобии,\n");
    printf("пожалуйста, введите секретный код: ");
    scanf("%d", &code_entered);
    while (code_entered != secret_code)
    {
        printf("Чтобы войти в клуб лечения трискадекафобии,\n");
        printf("пожалуйста, введите секретный код: ");
        scanf("%d", &code_entered);
    }
    printf("Поздравляем! Вас вылечили!\n");

    return 0;
}
```

Общая форма цикла do while имеет вид:

```
do
    оператор
while ( выражение );
```

Оператор может быть простым или составным. Обратите внимание на то, что сам цикл do while считается оператором и таким образом требует наличия после него точки с запятой (рис. 6.5).

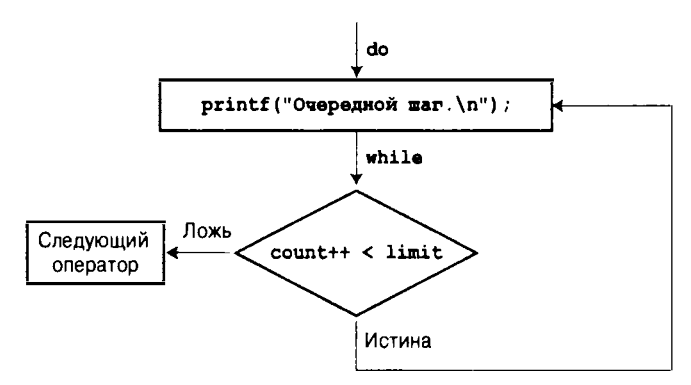


Рис. 6.5. Структура цикла `do while`

Цикл `do while` всегда выполняется, по меньшей мере, один раз, потому что проверка условия производится после того, как тело цикла выполнено. С другой стороны, циклы `for` и `while` могут не выполниться ни разу, поскольку проверка условия цикла осуществляется перед входом в цикл. Использование циклов `do while` должно быть ограничено случаями, при которых требуется выполнение хотя бы одной итерации. Например, программа запроса пароля мола бы содержать цикл, как демонстрируется в следующем псевдокоде:

```
do
{
   запросить ввод пароля
   прочитать пользовательский ввод
} while (введенные данные не совпадают с паролем);
```

Избегайте применения структуры `do while`, которая имеет вид, подобный показанному ниже псевдокоду:

```
do
{
   запросить у пользователя, желает ли он продолжить
   какие-то действия
} while (ответом является 'да');
```

В этом случае, даже если пользователь ответит "нет" на запрос о продолжении, действия все равно выполняются, т.к. проверка введенного ответа происходит слишком поздно.

Сводка: оператор `do while`

Ключевые слова

`do while`

Общий комментарий

Оператор `do while` создает цикл, который повторяется до тех пор, пока проверочное *выражение* не станет ложным или нулевым. Оператор `do while` является циклом с постусловием, т.е. решение о следующем проходе цикла принимается после выполнения текущей итерации. Таким образом, цикл должен быть выполнен, по меньшей мере, один раз. Часть *оператор* может быть простым или составным оператором.

Форма записи

```
do
   оператор
while (выражение);
```

Часть *оператор* повторяется до тех пор, пока *выражение* не станет ложным или нулевым.

Пример:

```
do
   scanf("%d", &number);
while (number !< 20)
```

Выбор подходящего цикла

Когда решено, что цикл необходим, возникает вопрос: какой из них должен использоваться? Для начала определитесь с тем, какого вида нужен цикл – с предусловием или с постусловием. Обычно выбирается цикл с предусловием. Исследователи в области вычислительной техники считают циклы с предусловием более удачными по нескольким причинам. Во-первых, в общем случае условие лучше проверять до выполнения итерации, а не после нее. Во-вторых, программа проще для восприятия, если условие цикла находится в начале цикла. В-третьих, во многих ситуациях важно, чтобы цикл был пропущен полностью, если условие изначально не удовлетворяется.

Предположим, что требуется цикл с предусловием. Это должен быть цикл `for` или же цикл `while`? Частично это дело вкуса, поскольку то, что возможно сделать с помощью одного цикла, можно достичь и посредством другого. Чтобы сделать цикл `for` похожим на `while`, можно не указывать первое и третье выражения. Так, цикл

```
for ( ; условие-проверки ; )
```

эквивалентен циклу

```
while (условие-проверки)
```

Чтобы придать циклу `while` вид, подобный `for`, предварите его инициализацией и предусмотрите внутри тела операторы для обновления значений. Например:

```
инициализация;
while (условие-проверки)
{
   тело-цикла;
   обновление;
}
```

эквивалентно

```
for (инициализация; условие-проверки; обновление)
   тело-цикла;
```

С позиций преобладающего стиля цикл `for` больше подходит в ситуациях, когда цикл предусматривает инициализацию и обновление переменной, а цикл `while` предпочтительнее, когда этого делать не нужно. Цикл `while` целесообразно применять для следующего условия:

```
while (scanf("%ld", &num) == 1)
```

Цикл `for` является более естественным выбором, когда реализуется подсчет для какого-нибудь индекса:

```
for (count = 1; count <= 100; count++)
```

Вложенные циклы

Вложенный цикл — это цикл внутри другого цикла. Вложенные циклы часто используются для отображения данных в виде строк и столбцов. Один цикл может обрабатывать, скажем, все столбцы в строке, а второй цикл — все строки. В листинге 6.17 приведен простой пример.

Листинг 6.17. Программа rows1.c

```
/* rows1.c -- применение вложенных циклов */
#include <stdio.h>
#define ROWS 6
#define CHARS 10
int main(void)
{
   int row;
   char ch;

   for (row = 0; row < ROWS; row++)              /* строка 10 */
   {
      for (ch = 'A'; ch < ('A' + CHARS); ch++)    /* строка 12 */
         printf("%c", ch);
      printf("\n");
   }

   return 0;
}
```

Выполнение этой программы дает следующий вывод:

```
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
```

Анализ программы

Цикл for, который начинается в строке 10, называется *внешним*, а цикл, начинающийся в строке 12 — *внутренним*. Внешний цикл стартует при значении 0 переменной row и завершается, когда ее значение достигает 6. Таким образом, происходит шесть итераций внешнего цикла, и переменная row последовательно получает значения от 0 до 5. Первый оператор в каждой итерации является внутренним циклом for. Он выполняет 10 итераций, выводя символы от A до J в одной строке. Второй оператор внешнего цикла, printf("\n");, начинает новую строку, так что при следующем выполнении внутреннего цикла вывод будет производиться в новой строке.

Будучи вложенным циклом, внутренний цикл проходит через полный диапазон итераций для каждой итерации внешнего цикла. В последнем примере внутренний цикл выводит 10 символов в строке, а внешний цикл создает 6 таких строк.

Изменение поведения вложенного цикла

В предшествующем примере внутренний цикл выполнял одни и те же действия для каждой итерации внешнего цикла. Однако можно сделать так, чтобы внутренний цикл вел себя по-разному в зависимости от внешнего цикла.

Например, в листинге 6.18 показан слегка измененный код предыдущей программы, в котором начальный символ внутреннего цикла изменяется в зависимости от номера итерации внешнего цикла. Вдобавок здесь используется комментарий более нового стиля и `const` вместо `#define`, что способствует освоению обоих подходов.

Листинг 6.18. Программа `rows2.c`

```
// rows2.c -- применение зависимых вложенных циклов
#include <stdio.h>
int main(void)
{
   const int ROWS = 6;
   const int CHARS = 6;
   int row;
   char ch;

   for (row = 0; row < ROWS; row++)
   {
      for (ch = ('A' + row); ch < ('A' + CHARS); ch++)
         printf("%c", ch);
      printf("\n");
   }

   return 0;
}
```

На этот раз вывод имеет такой вид:

```
ABCDEF
BCDEF
CDEF
DEF
EF
F
```

Поскольку на каждой итерации внешнего цикла `row` добавляется к `'A'`, переменная `ch` в каждой строке инициализируется следующим по порядку символом. Тем не менее, условие проверки не изменялось, так что каждая новая строка по-прежнему заканчивается символом `F`. В результате каждая следующая выводимая строка содержит на один символ меньше, чем предыдущая.

Введение в массивы

Массивы являются важным инструментом во многих программах. Они позволяют хранить несколько элементов связанной информации в удобной форме. Массивам полностью посвящена глава 10, но поскольку массивы часто используются с циклами, мы представим их прямо сейчас.

Массив — это совокупность значений одного и того же типа, такая как 10 значений `char` или 15 значений `int`, которые хранятся в памяти последовательно. Массив целиком носит свое имя, а доступ к его отдельным *элементам* осуществляется с применением целочисленного индекса. Например, объявление

```
float debts[20];
```

сообщает о том, что `debts` является массивом с 20 элементами, каждый из которых может содержать в себе значение `float`. Первый элемент массива называется `debts[0]`, второй элемент — `debts[1]` и т.д. вплоть до `debts[19]`.

Обратите внимание, что нумерация элементов массива начинается с 0, а не с 1. Каждому элементу массива может быть присвоено значение float. К примеру, можно записать следующий код:

```
debts[5] = 32.54;
debts[6] = 1.2e+21;
```

В сущности, элемент массива можно использовать тем же самым образом, как это делалось бы с переменной такого же типа. Например, можно прочитать значение и поместить его в конкретный элемент:

```
scanf("%f", &debts[4]);    // чтение значения в 5-й элемент массива
```

Потенциальная ловушка здесь в том, что в интересах скорости вычислений корректность указанного индекса не проверяется. Ниже приведены примеры ошибочных операторов:

```
debts[20] = 88.32;        // такой элемент массива не существует
debts[33] = 828.12;       // такой элемент массива не существует
```

Тем не менее, компилятор не обнаруживает ошибки подобного рода. Во время выполнения программы эти операторы поместили бы данные в ячейки памяти, которые возможно заняты другими данными, потенциально искажая вывод программы или даже приводя к ее аварийному завершению.

Массив может относиться к любому типу данных:

```
int nannies[22];          /* массив для хранения 22 целых чисел          */
char actors[26];          /* массив для хранения 26 символов             */
long big[500];            /* массив для хранения 500 целых чисел типа long */
```

Ранее в книге мы обсуждали строки, которые представляют собой специальный случай того, что можно хранить в массиве типа char. (В общем случае массив типа char содержит элементы, которым присваиваются значения char.) Содержимое массива char формирует строку, если массив содержит нулевой символ (\0), обозначающий конец строки (рис. 6.6).

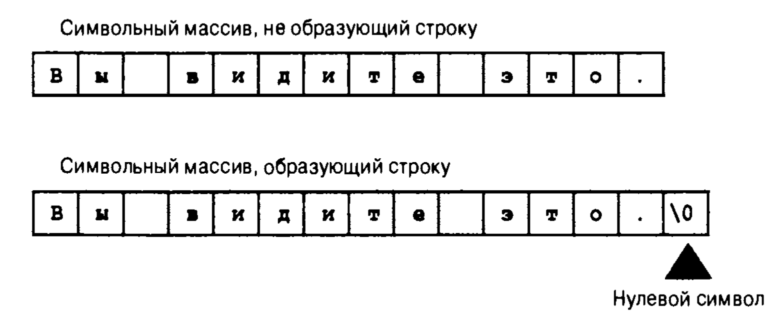


Рис. 6.6. Символьные массивы и строки

Числа, применяемые для идентификации элементов массива, называются *индексами* или *смещениями*. Индексы должны быть целыми числами, к тому же, как было указано ранее, индексация начинается с 0. Элементы массива хранятся в памяти рядом друг с другом (рис. 6.7).

int boo[4] (2 байта на значение типа **int**)

1980	46	4816	3
boo[0]	**boo[1]**	**boo[2]**	**boo[3]**

char foo[4] (1 байт на значение **char**)

в	**о**	**д**	**а**
foo[0]	**foo[1]**	**foo[2]**	**foo[3]**

Рис. 6.7. Массивы char и int в памяти

Использование цикла for с массивами

Массивы применяются в огромном числе ситуаций. В листинге 6.19 демонстрируется относительно простой случай. Эта программа считывает 10 результатов игры в гольф, которые позже будут обрабатываться. За счет использования массива вы избегаете необходимости объявлять 10 переменных с разными именами, по одной для каждого результата. Кроме того, для чтения входных данных можно применять цикл for. Программа предназначена для подсчета общей суммы результатов, их среднего значения и гандикапа, который представляет собой разность между средним и стандартным результатом, или *па*ром. (Пар – это термин в гольфе, означающий количество ударов по мячу, которое необходимо опытному игроку для проведения мяча в лунку или прохода всех лунок – *прим.перев.*)

Листинг 6.19. Программа scores_in.c

```
// scores_in.c -- использование циклов для обработки массивов
#include <stdio.h>
#define SIZE 10
#define PAR 72
int main(void)
{
   int index, score[SIZE];
   int sum = 0;
   float average;

   printf("Введите %d результатов игры в гольф:\n", SIZE);
   for (index = 0; index < SIZE; index++)
      scanf("%d", &score[index]);          // чтение 10 результатов игры в гольф
   printf("Введены следующие результаты:\n");
   for (index = 0; index < SIZE; index++)
      printf("%5d", score[index]);         // проверка введенной информации
   printf("\n");
   for (index = 0; index < SIZE; index++)
      sum += score[index];                 // сложение результатов
   average = (float) sum / SIZE;           // проверенный временем метод
   printf("Сумма результатов = %d, среднее значение = %.2f\n", sum, average);
   printf("Полученный гандикап равен %.0f.\n", average - PAR);

   return 0;
}
```

Давайте посмотрим, работает ли программа из листинга 6.19, а после этого прокомментируем ее действия. Ниже показан вывод программы:

```
Введите 10 результатов игры в гольф:
99 95 109 105 100
96 98 93 99 97 98
Введены следующие результаты:
  99  95  109  105  100   96   98   93   99   97
Сумма результатов = 991, среднее значение = 99.10
Полученный гандикап равен 27.
```

Программа работает, поэтому приступим к исследованию деталей. Прежде всего, обратите внимание, что хотя в примере было набрано 11 чисел, прочитались только 10 из них, т.к. цикл чтения считывает только 10 значений. Поскольку функция `scanf()` пропускает пробельные символы, можно вводить в одной строке все 10 чисел, вводить каждое число в отдельной строке либо, как в рассматриваемом случае, для разделения значений воспользоваться смесью символов новой строки и пробела. (Из-за буферизации ввода числа передаются в программу только после нажатия клавиши <Enter>.)

Работать с массивами и циклами гораздо удобнее, чем применять 10 операторов `scanf()` и 10 операторов `printf()` для чтения и вывода 10 результатов. Цикл `for` предлагает простой и прямолинейный способ использования индексов массива. Следует отметить, что элемент массива `int` обрабатывается подобно переменной типа `int`. Для чтения переменной `fue` типа `int` применялся бы вызов `scanf("%d", &fue)`. В листинге 6.19 считывается элемент `score[index]` типа `int`, поэтому используется вызов `scanf("%d", &score[index])`.

В данном примере проиллюстрировано несколько аспектов, касающихся стиля программирования. Во-первых, применение директивы `#define` для создания символической константы (`SIZE`), указывающей размер массива, является хорошей идеей. Эта константа используется в определении массива и при установке пределов в циклах. Если позже понадобится расширить программу для обработки 20 результатов, достаточно просто переопределить константу `SIZE`, сделав ее равной 20. Вам не придется изменять каждую часть программы, в которой участвует размер массива. Во-вторых, конструкция

```
for (index = 0; index < SIZE; index++)
```

удобна для обработки массива с размером `SIZE`. Очень важно указывать правильные пределы массива. Первый элемент имеет индекс `0`, и цикл начинается с установки `index` в `0`. Поскольку нумерация начинается с `0`, индексом последнего элемента является `SIZE - 1`. То есть десятый элемент массива — это `score[9]`. Условие проверки `index < SIZE` обеспечивает это, делая последним применяемым значением `index` величину `SIZE - 1`.

В-третьих, в программах рекомендуется выводить на экран значения, которые были только что прочитаны (для эхо-контроля). Это помогает удостовериться в том, что программа обрабатывает именно те данные, которые ожидаются.

Наконец, в-четвертых, обратите внимание на использование в листинге 6.19 трех отдельных циклов `for`. Вас может интересовать, действительно ли это необходимо. Можно ли объединить некоторые из операций в один цикл? Да, вы могли бы поступить так. Программа стала бы компактнее. Однако это противоречило бы принципу *модульности*. Идея, лежащая в основе этого термина, заключается в том, что программа должна быть разбита на отдельные модули, и каждый модуль должен решать одну задачу. Это облегчает чтение программы. Но что вероятно даже важнее — модульность

намного упрощает обновление или модификацию программы, потому что ее части не перемешаны. Когда вы обретете достаточные знания о функциях, то сможете поместить каждый модуль в функцию, улучшая модульность программы.

Пример цикла, использующего возвращаемое значение функции

В последнем примере этой главы применяется функция, которая вычисляет результат возведения числа в целую степень. (Для решения более сложных числовых задач в библиотеке math.h предлагается более мощная функция pow(), которая позволяет указывать степени с плавающей запятой.) Тремя главными задачами, которые решаются в этом упражнении, являются: разработка алгоритма вычисления ответа, представление этого алгоритма в виде функции, возвращающей ответ, и предоставление удобного способа тестирования этой функции.

Сначала давайте обратимся к алгоритму. Мы сохраним функцию простой, ограничив ее положительными целочисленными степенями. Тогда для возведения числа n в степень p переменную n нужно просто умножить на саму себя p раз. Это совершенно естественная задача для цикла. Вы можете установить переменную pow в 1, после чего многократно умножать ее на n:

```
for(i = 1; i <= p; i++)
    pow *= n;
```

Вспомните, что операция *= приводит к умножению левой части выражения на правую часть. После первой итерации цикла pow имеет значение 1, умноженное на n, т.е. n. После второй итерации переменная pow равна ее предыдущему значению (n), умноженному на n, или n в квадрате, и т.д. В этом контексте цикл for является вполне естественным, потому что цикл выполняется заранее известное (после того, как становится известным p) количество раз.

Теперь, когда построен алгоритм, мы можем решить, какие типы данных использовать. Показатель степени p, будучи целочисленным, должен иметь тип int. Чтобы обеспечить достаточный диапазон значений для переменной n и ее степеней, выберем для переменных n и pow тип double.

Давайте далее обдумаем, как оформить функцию. Нам необходимо предоставить функции два значения, а она должна возвратить одно значение. Чтобы передать функции необходимую информацию, можно определить два аргумента, один типа double и один типа int, указывающие число и степень. Как обеспечить возвращение функцией значения в вызывающую программу? Для написания функции с возвращаемым значением выполните следующие действия.

1. При определении функции установите тип значения, которое она возвращает.
2. С помощью ключевого слова return укажите возвращаемое значение.

Например, можно поступить следующим образом:

```
double power(double n, int p)  // возвращает тип double
{
   double pow = 1;
   int i;

   for (i = 1; i <= p; i++)
      pow *= n;
      return pow;             // возвратить значение переменной pow
}
```

Чтобы объявить возвращаемый тип функции, предварите этим типом имя функции, как это делается при объявлении переменной. Ключевое слово `return` заставляет функцию возвратить в вызывающую функцию значение, указанное после `return`. Здесь функция возвращает значение переменной, но она может также возвращать значения выражений. Например, следующий оператор вполне допустим:

```
return 2 * x + b;
```

Функция вычислит значение выражения и возвратит его. В вызывающей функции возвращаемое значение может быть присвоено другой переменной, использовано как значение выражения, передано в качестве аргумента другой функции (как в `printf("%f", power(6.28, 3))`) или просто проигнорировано.

Теперь давайте применим функцию в программе. При тестировании функции было бы удобно располагать возможностью передачи этой функции нескольких значений, чтобы увидеть, как она реагирует. Это предполагает настройку цикла ввода. Естественным вариантом является цикл `while`. Функцию `scanf()` можно использовать для ввода двух значений одновременно. Если функция `scanf()` успешно прочитает два значения, она возвратит 2, поэтому циклом можно управлять, сравнивая возвращаемое значение `scanf()` со значением 2. Еще один момент: чтобы воспользоваться функцией `power()` в программе, ее понадобится объявить, как вы объявляете применяемые в программе переменные. Окончательная программа приведена в листинге 6.20.

Листинг 6.20. Программа `power.c`

```
// power.c -- возводит числа в целые степени
#include <stdio.h>
double power(double n, int p);      // прототип ANSI
int main(void)
{
   double x, xpow;
   int exp;

   printf("Введите число и положительную целую степень,");
   printf(" в которую\nчисло будет возведено. Для завершения программы");
   printf(" введите q.\n");
   while (scanf("%lf%d", &x, &exp) == 2)
   {
      xpow = power(x,exp);          // вызов функции
      printf("%.3g в степени %d равно %.5g\n", x, exp, xpow);
      printf("Введите следующую пару чисел или q для завершения.\n");
   }
   printf("Надеемся, что вы оценили это упражнение -- до свидания!\n");

   return 0;
}

double power(double n, int p)       // определение функции
{
   double pow = 1;
   int i;

   for (i = 1; i <= p; i++)
      pow *= n;

   return pow;                      // возвратить значение pow
}
```

Ниже показан пример выполнения этой программы:

```
Введите число и положительную целую степень, в которую
число будет возведено. Для завершения программы введите q.
1.2 12
1.2 в степени 12 равно 8.9161
Введите следующую пару чисел или q для завершения.
2
16
2 в степени 16 равно 65536
Введите следующую пару чисел или q для завершения.
q
Надеемся, что вы оценили это упражнение -- до свидания!
```

Анализ программы

Программа main() представляет собой пример *драйвера* — короткой программы, предназначенной для тестирования функции.

Цикл while является обобщением формы, используемой ранее. Ввод 1.2 12 приводит к успешному чтению функцией scanf() двух значений и возвращению 2, после чего цикл продолжается. Поскольку scanf() игнорирует пробельные символы, входные данные можно разносить по нескольким строкам, как демонстрируется в показанном выводе, но ввод q дает возвращаемое значение 0, потому что символ q не может быть прочитан, учитывая указанный спецификатор %lf. Это приводит к тому, что scanf() возвращает 0, тем самым прекращая выполнение цикла. Аналогично, ввод 2.8 q вызвал бы возврат функцией scanf() значения 1, что также бы обеспечило завершение цикла.

Теперь рассмотрим все детали, связанные с функцией. Функция power() появляется в программе трижды. Первый раз она встречается в следующей конструкции:

```
double power(double n, int p);   // прототип ANSI
```

Этот оператор сообщает, или *объявляет*, что в программе будет применяться функция по имени power(). Первое ключевое слово double отражает, что power() возвращает значение типа double. Компилятор должен знать вид значения, возвращаемого power(), чтобы определить, сколько байтов данных следует ожидать и как их интерпретировать; это и является причиной объявления функции. Конструкция double n, int p внутри круглых скобок означает, что функция power() принимает два аргумента. Первый аргумент должен быть значением типа double, а второй — значением типа int.

Второй раз функция появляется в следующем операторе:

```
xpow = power(x,exp);             // вызов функции
```

В этом месте программа вызывает функцию и передает ей два значения. Функция вычисляет значение x в степени exp и возвращает результат в вызывающую программу, где возвращаемое значение присваивается переменной xpow.

Третий раз рассматриваемая функция встречается в заголовке определения функции:

```
double power(double n, int p)    // определение функции
```

Здесь функция power() принимает два аргумента, double и int, представленные переменными n и p. Обратите внимание, что в определении функции точка с запятой после power() отсутствует, но в объявлении функции она имеется. После заголовка следует код, который указывает, что делает функция power().

Вспомните, что в функции используется цикл for для вычисления значения n в степени p, которое затем присваивается переменной pow. Следующая строка делает это значение pow возвращаемым значением функции:

```
return pow;                    // возвратить значение pow
```

Использование функций с возвращаемыми значениями

Объявление, вызов и определение функции, а также применение ключевого слова return — все это базовые элементы в определении и использовании функции с возвращаемым значением.

К настоящему времени у вас могли накопиться вопросы. Например, если вы обязаны объявлять функции до применения их возвращаемых значений, то каким образом получилось воспользоваться возвращаемым значением функции scanf() без ее объявления? Почему вы должны объявить функцию power() отдельно, если в ее определении указано, что она возвращает тип double?

Рассмотрим сначала второй вопрос. Компилятор должен знать, какой тип возвращает функция power(), когда он впервые сталкивается с ней в программе. К этому моменту компилятор еще не встретил определение power(), поэтому ему пока не известно, что в определении данной функции указан возвращаемый тип double. Чтобы помочь компилятору, вы используете *предварительное объявление*. Это объявление информирует компилятор о том, что функция power() объявлена где-то в другом месте, и она будет возвращать тип double. Если поместить определение функции power() раньше функции main() внутри файла, то предварительное объявление можно не указывать, т.к. компилятор будет располагать всеми сведениями о функции power() до достижения функции main(). Тем не менее, это не стандартный стиль программирования на C. Поскольку функция main() обычно предоставляет базовую структуру для программы, лучше размещать ее код первой. Кроме того, функции часто определяются в отдельных файлах, и в этих случаях предварительные объявления обязательны.

А почему мы не объявили функцию scanf()? На самом деле мы сделали это. В заголовочном файле stdio.h содержатся объявления scanf(), printf() и других функций ввода-вывода. В объявлении функции scanf() указано, что она возвращает значение типа int.

Ключевые понятия

Цикл представляет собой мощный инструмент программирования. При написании цикла вы должны обращать особое внимание на следующие три аспекта.

- Четкое определение условия прекращения цикла.
- Обеспечение инициализации значений, задействованных в условии проверки цикла, перед первым их использованием.
- Обеспечение в цикле действий по обновлению условия проверки на каждой итерации.

Условия проверки обрабатываются путем их числовой оценки. Результат, равный 0, трактуется как ложное значение, а любое другое числовое значение — как истинное. Выражения с операциями отношений часто выступают в качестве условий проверки и являются чуть более специфичными. Результатом такого выражения будет 1, если оно истинно, и 0, если ложно, что соответствует значениям, разрешенным для нового типа _Bool.

Массивы состоят из расположенных рядом ячеек памяти со значениями одного и того же типа. Вы должны помнить, что нумерация элементов массива начинается с 0, поэтому последний элемент массива имеет индекс на единицу меньше количества элементов. В С не выполняется проверка допустимости значений индексов, так что ответственность за это возлагается целиком на вас.

Использование функций включает три отдельных шага.

1. Объявление функции посредством ее прототипа.
2. Выполнение функции внутри программы путем ее вызова.
3. Определение функции.

Прототип позволяет компилятору проверять, корректно ли применяется функция, а определение функции указывает, как она должна работать. Прототип и определение функции являются примерами современного стиля программирования, предусматривающего разделение элемента программы на интерфейс и реализацию. Интерфейс описывает, как используется средство, что и делает прототип, а реализация далее расписывает конкретные действия, чем занимается определение функции.

Резюме

Главной темой этой главы было управление выполнением программы. Язык С предлагает много средств для структурирования программ. Операторы `while` и `for` позволяют строить циклы с предусловием. Оператор `for` особенно хорошо подходит для циклов, в которых производится инициализация и обновление. Инициализировать и обновлять более одной переменной в цикле `for` можно с помощью операции запятой. Для менее распространенных случаев, когда требуется цикл с постусловием, в языке С предусмотрен оператор `do while`.

Типичная конструкция цикла `while` имеет следующий вид:

```
получить первое значение
while (значение удовлетворяет условию проверки)
{
   обработать значение
   получить следующее значение
}
```

Цикл `for`, выполняющий те же действия, выглядит так:

```
for (получить первое значение; значение удовлетворяет условию проверки;
получить следующее значение)
   обработать значение
```

Во всех этих циклах условие проверки служит для выяснения, должна ли выполняться еще одна итерация цикла. В общем случае цикл продолжает работу, если проверочное выражение имеет ненулевое значение, иначе цикл завершается. Часто условием проверки является выражение отношения, которое представляет собой выражение, содержащее операцию отношения. Такое выражение получает значение `1`, если отношение истинно, и `0` — во всех остальных случаях. Переменные типа `_Bool`, введенного стандартом С99, могут принимать только значения `1` и `0`, обозначающие "истину" и "ложь".

В дополнение к операциям отношений в главе рассматривались арифметические операции присваивания языка С, такие как += и *=. Эти операции модифицируют значения операнда слева от знака операции, выполняя над ним указанные арифметические операции.

Следующей темой были массивы. Массив объявляется с применением квадратн скобок для указания количества элементов в нем. Первый элемент произвольного м сива имеет номер 0, второй – номер 1 и т.д. Например, объявление

```
double hippos[20];
```

создает массив из 20 элементов; отдельные элементы массива получают имена в диа зоне от hippos[0] до hippos[19]. Манипулировать индексами, используемыми , нумерации элементов массива, удобно с помощью циклов. Наконец, в главе было пс зано, как создавать и выполнять функцию с возвращаемым значением.

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля приведены в приложении А.

1. Каким будет значение quack после выполнения каждой строки кода? Каждый последних пяти операторов использует значение quack, сгенерированное пр шествующим оператором.

```
int quack = 2;
quack += 5;
quack *= 10;
quack -= 6;
quack /= 8;
quack %= 3;
```

2. При условии, что переменная value имеет тип int, определите, какой вы будет получен в результате выполнения следующего цикла:

```
for ( value = 36; value > 0; value /= 2)
  printf("%3d", value);
```

Какие проблемы могли бы возникнуть, если бы переменная value имела т double вместо int?

3. Запишите выражение для каждого из следующих условий проверки.

а. x больше 5.

б. Функция scanf() предпринимает неудачную попытку прочитать одно зна ние типа double (по имени x).

в. x имеет значение 5.

4. Запишите выражение для каждого из следующих условий проверки.

а. Функция scanf() успешно читает одно целое число.

б. x не равно 5.

в. x равно 20 или больше.

5. Приведенная ниже программа не идеальна. Какие ошибки вы можете найти?

```
#include <stdio.h>
int main(void)
{                                 /* строка 3  */
   int i, j, list(10);            /* строка 4  */
   for (i = 1, i <= 10, i++)      /* строка 6  */
   {                              /* строка 7  */
      list[i] = 2*i + 3;          /* строка 8  */
      for (j = 1, j > = i, j++)   /* строка 9  */
         printf(" %d", list[j]);  /* строка 10 */
      printf("\n");               /* строка 11 */
   }                              /* строка 12 */
```

6. Воспользуйтесь вложенными циклами для написания программы, которая выводит следующую последовательность символов:

```
$$$$$$$$
$$$$$$$$
$$$$$$$$
$$$$$$$$
```

7. Что выведет каждая из следующих программ?

```
#include <stdio.h>

int main(void)
{
   int i = 0;

   while (++i < 4)
      printf("Hi! ");
   do
      printf("Bye! ");
   while (i++ < 8);
   return 0;
}
```

```
#include <stdio.h>

int main(void)
{
   int i;
   char ch;

   for (i = 0, ch = 'A'; i < 4; i++, ch += 2 * i)
        printf("%c", ch);
   return 0;
}
```

8. Что выведут приведенные далее программы в случае ввода `Go west, young man!`? (В кодировке ASCII символ `!` следует за символом пробела.)

```
#include <stdio.h>

int main(void)
{
   char ch;

   scanf("%c", &ch);
   while ( ch != 'g' )
   {
      printf("%c", ch);
      scanf("%c", &ch);
   }
   return 0;
}
```

```
#include <stdio.h>

int main(void)
{
   char ch;

   scanf("%c", &ch);
   while ( ch != 'g' )
   {
```

```
      printf("%c", ++ch);
      scanf("%c", &ch);
   }
   return 0;
}
```

```
#include <stdio.h>
int main(void)
{
   char ch;
   do {
      scanf("%c", &ch);
      printf("%c", ch);
   } while ( ch != 'g' );
   return 0;
}
```

```
#include <stdio.h>
int main(void)
{
   char ch;
   scanf("%c", &ch);
   for ( ch = '$'; ch != 'g'; scanf("%c", &ch) )
        printf("%c", ch);
   return 0;
}
```

9. Что выведет следующая программа?

```
#include <stdio.h>
int main(void)
{
   int n, m;
   n = 30;
   while (++n <= 33)
      printf("%d|",n);
      n = 30;
   do
      printf("%d|",n);
   while (++n <= 33);
   printf("\n***\n");
   for (n = 1; n*n < 200; n += 4)
      printf("%d\n", n);
      printf("\n***\n");
   for (n = 2, m = 6; n < m; n *= 2, m+= 2)
      printf("%d %d\n", n, m);
   printf("\n***\n");
   for (n = 5; n > 0; n--)
   {
      for (m = 0; m <= n; m++)
         printf("=");
      printf("\n");
   }
   return 0;
}
```

10. Взгляните на следующее объявление:

```
double mint[10];
```

а. Какое имя назначено массиву?

б. Сколько элементов в массиве?

в. Какие виды значений могут храниться в каждом элементе массива?

г. Что из перечисленного ниже является корректным использованием функции `scanf()` с этим массивом?

- `scanf("%lf",mint[2])`
- `scanf("%lf", &mint[2])`
- `scanf("%lf", &mint)`

11. Кое-кому нравится считать двойками, поэтому он написал программу, которая создает массив и заполняет его четными числами 2, 4, 6, 8 и т.д. Есть ли ошибки в этой программе?

```
#include <stdio.h>
#define SIZE 8
int main(void)
{
   int by_twos[SIZE];
   int index;

   for (index = 1; index <= SIZE; index++)
      by_twos[index] = 2 * index;
   for (index = 1; index <= SIZE; index++)
      printf("%d ", by_twos);
   printf("\n");
   return 0;
}
```

12. Вы хотите написать функцию, которая возвращает значение типа `long`. Что должно включать определение этой функции?

13. Определите функцию, которая принимает аргумент типа `int` и возвращает результат его возведения в квадрат как значение типа `long`.

14. Что выведет следующая программа?

```
#include <stdio.h>
int main(void)
{
   int k;
   for(k = 1, printf("%d: Hi!\n", k); printf("k = %d\n",k),
      k*k < 26; k+=2, printf("Now k is %d\n", k) )
         printf("k is %d in the loop\n",k);
   return 0;
}
```

Упражнения по программированию

1. Напишите программу, которая создает массив из 26 элементов и помещает в него 26 строчных букв английского алфавита. Также предусмотрите вывод содержимого этого массива.

2. Воспользуйтесь вложенными циклами, чтобы написать программу, которая выводит следующую последовательность символов:

```
$
$$
$$$
$$$$
$$$$$
```

3. Воспользуйтесь вложенными циклами, чтобы написать программу, которая выводит следующую последовательность символов:

```
F
FE
FED
FEDC
FEDCB
FEDCBA
```

Примечание: если в вашей системе не используется ASCII или какая-то другая кодировка, в которой буквы представлены в числовом порядке, то для инициализации символьного массива буквами алфавита вы можете применять следующее объявление:

```
char lets[27] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Затем для выбора конкретных букв можно использовать индексы массива, например, `lets[0]` для `'A'` и т.д.

4. Воспользуйтесь вложенными циклами, чтобы написать программу, которая выводит следующую последовательность символов:

```
A
BC
DEF
GHIJ
KLMNO
PQRSTU
```

Если в вашей системе не используется кодировка, в которой буквы представлены в числовом порядке, см. примечание в упражнении 3.

5. Напишите программу, которая предлагает пользователю ввести прописную букву. Воспользуйтесь вложенными циклами, чтобы написать программу, которая выводит фигуру в виде пирамиды, подобную приведенной ниже:

```
    A
   ABA
  ABCBA
 ABCDCBA
ABCDEDCBA
```

Эта фигура должна расширяться вплоть до введенного символа. Например, представленная фигура стала результатом ввода символа `E`. Подсказка: для обработки строк воспользуйтесь внешним циклом. Для формирования строки применяйте три внутренних цикла: один для манипуляции пробелами, второй для вывода букв в порядке возрастания и третий для вывода букв в порядке убывания. Если в вашей системе не используется ASCII или подобная ей кодировка, в которой буквы представлены в числовом порядке, см. примечание в упражнении 3.

6. Напишите программу для вывода таблицы, в каждой строке которой представлено целое число, его квадрат и его куб. Запросите у пользователя верхний и нижний пределы таблицы. Используйте цикл `for`.

7. Напишите программу, которая читает слово в символьный массив, а затем выводит это слово в обратном порядке. Подсказка: воспользуйтесь функцией `strlen()` (глава 4) для вычисления индекса последнего символа массива.

8. Напишите программу, которая запрашивает два числа с плавающей запятой и выводит значение их разности, деленной на их произведение. Программа должна обрабатывать пары вводимых чисел до тех пор, пока пользователь не введет нечисловое значение.

9. Модифицируйте упражнение 8 так, чтобы программа использовала функцию для возврата результатов вычислений.

10. Напишите программу, которая запрашивает ввод нижнего и верхнего пределов последовательности целых чисел, вычисляет сумму всех квадратов целых чисел, начиная с квадрата нижнего целочисленного предела и заканчивая квадратом верхнего целочисленного предела, после чего отображает результат на экране. Затем программа должна запрашивать ввод следующих предельных значений и отображать ответ, пока пользователь не введет значение верхнего предела, который меньше или равен нижнему пределу. Результаты выполнения программы должны выглядеть следующим образом:

    ```
    Введите нижний и верхний целочисленные пределы: 5 9
    Сумма квадратов целых чисел от 25 до 81 равна 255
    Введите следующую комбинацию пределов: 3 25
    Сумма квадратов целых чисел от 9 до 625 равна 95
    Введите следующую комбинацию пределов: 5 5
    Работа завершена
    ```

11. Напишите программу, которая читает восемь целых чисел в массив, после чего выводит их в обратном порядке.

12. Взгляните на следующие две бесконечных последовательности:

    ```
    1.0 + 1.0/2.0 + 1.0/3.0 + 1.0/4.0 +...
    1.0 - 1.0/2.0 + 1.0/3.0 - 1.0/4.0 +...
    ```

    Напишите программу, которая подсчитывает промежуточные суммы этих двух последовательностей до тех пор, пока не будет обработано заданное количество элементов. Подсказка: произведение нечетного количества значений -1 равно -1, а произведение четного количества значений -1 равно 1. Предоставьте пользователю возможность вводить предел интерактивно; нулевое или отрицательное значение должно завершать ввод. Просмотрите промежуточные суммы для 100, 1000 и 10,000 элементов. Сходятся ли эти последовательности к какому-то значению?

13. Напишите программу, которая создает восьмиэлементный массив типа `int` и помещает в него элементы начальных восьми степеней 2, а затем выводит полученные значения. Применяйте цикл `for` для вычисления элементов массива, и ради разнообразия для отображения значений воспользуйтесь циклом `do while`.

14. Напишите программу, которая создает два восьмиэлементных массива типа `double` и использует цикл для ввода значений восьми элементов первого массива. Программа должна накапливать в элементах второго массива суммы первого

массива с нарастающим итогом. Например, четвертый элемент второго массива должен быть равен сумме первых четырех элементов первого массива, а пятый элемент второго массива — сумме пяти первых элементов первого массива. (Это можно сделать с помощью вложенных циклов, однако если учесть тот факт, что пятый элемент второго массива равен четвертому элементу второго массива плюс пятый элемент первого массива, можно избежать вложенных циклов и применить для решения задачи единственный цикл.) В завершение воспользуйтесь циклом для вывода содержимого обоих массивов, при этом первый массив должен отображаться в первой строке, а каждый элемент второго массива должен помещаться непосредственно под соответствующим элементом первого массива.

15. Напишите программу, которая читает строку ввода, а затем выводит ее в обратном порядке. Ввод можно сохранять в массиве значений типа `char`; предполагается, что строка состоит не более чем из 255 символов. Вспомните, что для чтения символа за раз можно применять функцию `scanf()` со спецификатором `%c`, а при нажатии клавиши <Enter> генерируется символ новой строки (`\n`).

16. Дафна делает вклад в сумме $100 под простые 10%. (То есть ежегодный прирост вклада составляет 10% от первоначальной суммы.) Дейдра вкладывает $100 под сложные 5%. (Это значит, что ежегодное увеличение вклада составляет 5% от текущего баланса, включающего предыдущий прирост вклада.) Напишите программу, которая вычисляет, сколько нужно лет, чтобы сумма на счету Дейдры превзошла сумму на счету Дафны. Выведите также размеры обоих вкладов на тот момент.

17. Чаки Лаки выиграл миллион долларов (после уплаты всех налогов), который он поместил на счет со ставкой 8% годовых. В последний день каждого года Чаки снимает со счета по $100 000. Напишите программу, которая вычисляет, сколько лет пройдет до того, как на счету Чаки не останется денег.

18. Профессор Робинс присоединился к группе в социальной сети. Вначале у него было пять друзей. Он заметил, что количество его друзей увеличивалось следующим образом. По истечении первой недели один человек вышел из числа друзей, а количество друзей удвоилось. По истечении второй недели двое вышли из числа друзей, а количество друзей удвоилось. Выражаясь в общем, по истечении *N*-й недели *N* людей выходили из числа друзей, а количество друзей удваивалось. Напишите программу, которая вычисляет количество друзей в конце каждой недели. Программа должна продолжать подсчет до тех пор, пока количество друзей не превысит число Данбара. Число Данбара — это приблизительная оценка максимального размера сплоченной социальной группы, в которой каждый член знает всех остальных членов и осведомлен об их взаимоотношениях друг с другом. Его приближенное значение равно 150.

7

Управляющие операторы C: ветвление и переходы

В ЭТОЙ ГЛАВЕ...

- Ключевые слова: if, else, switch, continue, break, case,default, goto
- Операции: && || ?:
- Функции: getchar(), putchar(), семейство ctype.h
- Использование операторов if и if else и вложение их друг в друга
- Использование логических операций для комбинирования выражений отношения в более сложные проверочные выражения
- Условные операции
- Оператор switch
- Операторы переходов break, continue и goto
- Использование функций символьного ввода-вывода: getchar() и putchar()
- Семейство функций анализа символов, предоставляемых заголовочным файлом ctype.h

По мере освоения языка С, скорее всего, вы захотите решать более сложные задачи. Тогда вам понадобятся способы управления и организации своих проектов. Для этого в С имеются необходимые инструментальные средства. Вы уже научились пользоваться циклами для программирования повторяющихся действий. В этой главе вы узнаете о структурах ветвления, таких как операторы if и switch, которые позволяют программе основывать свои действия на условиях проверки. Кроме того, вы получите начальные сведения о логических операциях С, позволяющих проверять более одного отношения в условиях while или if, а также ознакомитесь с операторами переходов, с помощью которых можно передавать управление в другую точку программы. К концу главы вы будете располагать всей базовой информацией, необходимой для проектирования программы, которая ведет себя желаемым образом.

Оператор if

Давайте начнем с простого примера применения оператора if, представленного в листинге 7.1. Эта программа считывает список показаний минимальных дневных температур (по шкале Цельсия) и выводит общее количество элементов, а также процент значений, меньших точки замерзания (т.е. ниже нуля по Цельсию). Для чтения значений используется функция scanf() в цикле. На каждой итерации цикла инкрементируется счетчик, отслеживая количество показаний. С помощью оператора if идентифицируются значения температуры ниже нуля, ведется отдельный подсчет таких случаев.

Листинг 7.1. Программа colddays.c

```
// colddays.c -- вычисляет процент случаев, когда температура опускается ниже нуля
#include <stdio.h>
int main(void)
{
   const int FREEZING = 0;
   float temperature;
   int cold_days = 0;
   int all_days = 0;
   printf("Введите список минимальных дневных температур.\n");
   printf("Используйте шкалу Цельсия; для завершения введите q.\n");
   while (scanf("%f", &temperature) == 1)
   {
      all_days++;
      if (temperature < FREEZING)
         cold_days++;
   }
   if (all_days != 0)
      printf("%d - общее количество дней: %.1f%% с температурой ниже нуля.\n",
         all_days, 100.0 * (float) cold_days / all_days);
   if (all_days == 0)
      printf("Данные не введены!\n");
      return 0;
}
```

Ниже показан результат выполнения этой программы:

```
Введите список минимальных дневных температур.
Используйте шкалу Цельсия, для завершения введите q.
12 5 -2.5 0 6 8 -3 -10 5 10 q
10 - общее количество дней: 30.0% дней с температурой ниже нуля.
```

В условии проверки цикла while значение, возвращаемое функцией scanf(), используется для завершения цикла, когда встречается нечисловое значение. За счет применения типа float вместо int для переменной temperature программа получает возможность принимать такие показания температуры, как -2.5, а также 8.

Вот новый оператор в блоке while:

```
if (temperature < FREEZING)
   cold_days++;
```

Этот оператор if инструктирует компьютер увеличить значение cold_days на 1, *если* только что считанное значение (temperature) меньше нуля. Что произойдет, если значение temperature не меньше нуля? Тогда оператор cold_days++; пропускается, а выполнение цикла while продолжается и читается следующее значение температуры. Оператор if еще два раза используется в программе для управления выводом. Если данные в наличии, программа выводит результаты. Если данные отсутствуют, программа сообщает об этом. (Вскоре мы рассмотрим более элегантный способ реализации этой части программы.)

Чтобы избежать целочисленного деления при вычислении процентного отношения, в примере выполняется приведение к типу float. На самом деле в этом приведении нет необходимости, т.к. входящее в выражение 100.0 * cold_days / all_days подвыражение 100.0 * cold_days вычисляется первым и принудительно приводится к типу с плавающей запятой правилами автоматического преобразования типов. Тем не менее, явное указание приведения типа документирует ваше намерение и помогает защитить программу от ошибочных переделок. Оператор if называется *оператором ветвления* или *оператором выбора*, потому что он представляет собой узловой пункт, где программа должна выбрать один из двух путей для дальнейшего следования. Общая форма оператора if имеет вид:

```
if (выражение)
   оператор
```

Если *выражение* имеет истинное (ненулевое) значение, то *оператор* выполняется. В противном случае он пропускается. Как и в цикле while, *оператор* может быть как одиночным, так и составным оператором. Его структура очень похожа на структуру while. Основное различие заключается в том, что в операторе if проверка условия и (возможное) выполнение производится всего лишь один раз, в то время как в цикле while проверка условия и выполнение могут повторяться многократно.

Обычно *выражение* является выражением отношения, т.е. в нем сравниваются две количественных величины, как в выражениях x > y и c == 6. Если *выражение* истинно (x больше y либо c равно 6), оператор выполняется, иначе оператор игнорируется. В общем случае можно применять любое выражение, при этом выражение, принимающее значение 0, трактуется как ложное.

Часть *оператор* может быть простым либо составным оператором или блоком, заключенным в фигурные скобки:

```
if (score > big)
   printf("Джекпот!\n");       // простой оператор
if (joe > ron)
{                              // составной оператор
   joecash++;
   printf("Ты проиграл, Ron.\n");
}
```

Обратите внимание, что вся структура if считается одним оператором, даже если в ней присутствует составной оператор.

Добавление к оператору `if` конструкции `else`

Простая форма оператора `if` предоставляет выбор между выполнением оператора (возможно, составного) и пропуском его. Язык С также позволяет выбирать один из двух операторов с использованием формы `if else`. Давайте применим форму `if else`, чтобы улучшить показанный ниже неуклюжий фрагмент кода из листинга 7.1:

```
if (all_days != 0)
   printf("%d - общее количество дней: %.1f%% с температурой ниже нуля.\n",
          all_days, 100.0 * (float) cold_days / all_days);
if (all_days == 0)
   printf("Данные не введены!\n");
```

Когда проверка значения `all_days` на неравенство 0 не проходит, то и без повторной проверки должно быть ясно, что оно равно нулю. Форма `if else` позволяет воспользоваться преимуществом этого знания, переписав данный фрагмент следующим образом:

```
if (all_days!= 0)
   printf("%d - общее количество дней: %.1f%% с температурой ниже нуля.\n",
          all_days, 100.0 * (float) cold_days / all_days);
else
   printf("Данные не введены!\n");
```

Здесь выполняется только одна проверка. Если проверочное выражение оператора `if` истинно, данные о температуре выводятся. Если же оно ложно, выводится предупреждающее сообщение. Вот общая форма оператора `if else`:

```
if (выражение)
   оператор1
else
   оператор2
```

Если *выражение* истинно (не равно нулю), выполняется *оператор1*. Если выражение ложно (равно нулю), выполняется оператор, следующий за `else`. Операторы могут быть простыми либо составными. Отступы в С не являются обязательными, но это стандартный стиль записи. Отступы зрительно выделяют операторы, выполнение которых зависит от условия проверки. Если между `if` и `else` нужно поместить более одного оператора, необходимо указать фигурные скобки, чтобы создать единый блок. В приведенной ниже конструкции нарушается синтаксис языка С, поскольку компилятор ожидает встретить между `if` и `else` только один оператор (простой или составной):

```
if (x > 0)
   printf("Инкрементирование x:\n");
   x++;
else         // здесь возникнет ошибка
   printf("x <= 0 \n");
```

Компилятор трактует оператор `printf()` как часть оператора `if`, но оператор `x++;` — как отдельный оператор, а не часть `if`. Поэтому компилятор считает, что `else` не принадлежит `if`, что является ошибкой. Взамен воспользуйтесь следующей формой:

```
if (x > 0)
{
   printf("Инкрементирование x:\n");
   x++;
}
else
   printf("x <= 0 \n");
```

Оператор `if` позволяет выбрать между выполнением или не выполнением одного действия. Оператор `if else` позволяет делать выбор между двумя действиями. На рис. 7.1 приведено сравнение этих двух операторов.

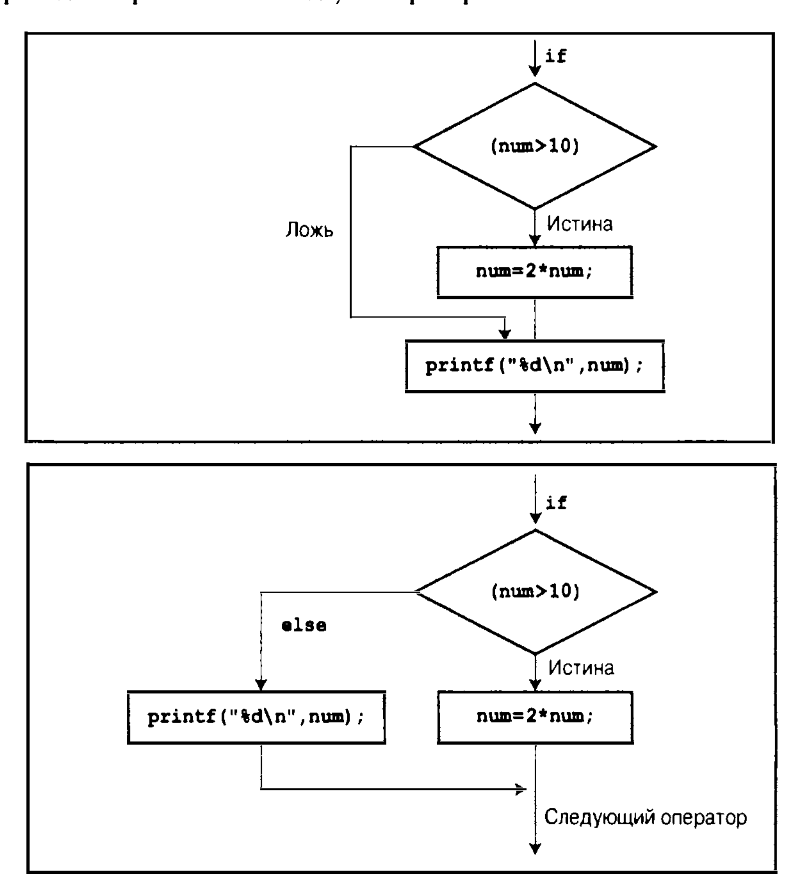


Рис. 7.1. Операторы `if` *и* `if else`

Еще один пример: знакомство с функциями getchar() и putchar()

В большинстве рассмотренных до сих пор примеров применялись числовые входные данные. Чтобы попрактиковаться с данными других типов, рассмотрим пример, ориентированный на обработку символов. Вы уже знаете, как использовать функции `scanf()` и `printf()` со спецификатором `%c` для чтения и вывода символов, но в этом примере мы будем иметь дело с парой функций C, специально предназначенных для символьного ввода-вывода — `getchar()` и `putchar()`.

Функция `getchar()` не принимает аргументов и возвращает очередной символ из входного потока. Например, показанный далее оператор читает следующий входной символ и присваивает его значение переменной `ch`:

```
ch = getchar();
```

Этот оператор дает тот же самый результат, что и оператор

```
scanf("%c", &ch);
```

Функция putchar() выводит переданный ей аргумент. Например, следующий оператор выводит в виде символа значение, ранее присвоенное переменной ch:

```
putchar(ch);
```

Этот оператор обеспечивает такой же результат, что и оператор

```
printf("%c", ch);
```

Поскольку эти функции работают только с символами, они быстрее и компактнее, чем более универсальные функции scanf() и printf(). Кроме того, обратите внимание, что они не нуждаются в спецификаторах формата, т.к. предназначены для работы только с символами. Обе функции обычно определены в файле stdio.h. (Кроме того, они обычно являются *макросами* препроцессора, а не истинными функциями; о функционально-подобных макросах речь пойдет в главе 16.)

Давайте посмотрим, как эти функции работают, написав программу, которая отображает введенную строку, но заменяет каждый отличный от пробела символ следующим за ним символом в последовательности кодов ASCII. Пробелы будут выводиться без изменений. Результат можно сформулировать так: "Если символ является пробелом, он выводится, в противном случае выводится символ, следующий за ним в последовательности кодов ASCII".

Код программы представлен в листинге 7.2.

Листинг 7.2. Программа cypher1.c

```
// cypher1.c -- вносит изменения во входные данные, сохраняя пробелы
#include <stdio.h>
#define SPACE ' '             // кавычка, пробел, кавычка
int main(void)
{
   char ch;

   ch = getchar();            // читать символ
   while (ch != '\n')         // пока не встретится конец строки
   {
      if (ch == SPACE)        // оставить пробел нетронутым
         putchar(ch);         // символ не меняется
      else
         putchar(ch + 1);     // изменить другие символы
      ch = getchar();         // получить следующий символ
   }
   putchar(ch);               // вывести символ новой строки

   return 0;
}
```

(Пусть вас не беспокоит предупреждение компилятора о возможной потере данных. Все прояснится в главе 8 при рассмотрении последовательности EOF.)

Ниже показан результат выполнения программы:

```
CALL ME HAL.
DBMM NF IBM/
```

Сравните этот цикл с циклом из листинга 7.1. В листинге 7.1 для определения момента прекращения цикла применялось возвращаемое значение scanf(), а не значение введенного элемента. Однако в листинге 7.2 для этой цели используется значение самого введенного элемента. Такое отличие в результате приводит к несколько другой структуре цикла, с одним оператором чтения перед циклом и еще одним оператором

чтения в конце цикла. Тем не менее, гибкий синтаксис языка С позволяет эмулировать программу из листинга 7.1 за счет объединения чтения и проверки в одно выражение. То есть цикл следующего вида:

```
ch = getchar();       /* читать символ                    */
while (ch != '\n')    /* пока не встретится конец строки  */
{
   ...                /* обработать символ                */
   ch = getchar();    /* получить следующий символ        */
}
```

можно заменить таким циклом:

```
while ((ch = getchar()) != '\n')
{
                      /* обработать символ                */
}
```

Интерес представляет следующая строка:

```
while ((ch = getchar()) != '\n')
```

Она демонстрирует характерный стиль программирования на языке С – объединение двух действий в одно выражение. Возможность свободного форматирования в С помогает сделать отдельные компоненты строки яснее:

```
while (
      (ch = getchar())              // присвоить значение переменной ch
                  != '\n')          // сравнить ch с \n
```

Действиями являются присваивание значения переменной ch и ее сравнение с символом новой строки. Круглые скобки, в которые заключено выражение ch = getchar(), делают его левым операндом операции !=. Чтобы вычислить это выражение, сначала вызывается функция getchar(), после чего возвращаемое ею значение присваивается переменной ch. Поскольку значением выражения присваивания является значение его левого члена, значением ch = getchar() будет как раз новое значение ch. Таким образом, после того как значение для ch прочитано, условие проверки сводится к ch != '\n' (т.е. значение ch *не* равно символу новой строки).

Конструкция подобного рода весьма распространена в программировании на С, поэтому вы должны быть с ней знакомы. Кроме того, не забывайте с помощью круглых скобок правильно группировать подвыражения.

Все скобки обязательны. Предположим, что вы по ошибке записали следующее выражение:

```
while (ch = getchar() != '\n')
```

Операция != имеет более высокий приоритет, чем =, следовательно, первым будет вычислено выражение getchar() != '\n'. Поскольку это условное выражение, оно принимает значение 1 или 0 (истина или ложь). Затем это значение присваивается переменной ch. Отсутствие скобок означает, что ch будет присвоено 0 или 1, а не возвращаемое значение функции getchar(); это совсем не то, что планировалось.

Оператор

```
putchar(ch + 1);  /* изменить другие символы */
```

еще раз иллюстрирует, что символы хранятся в виде целых чисел. В выражении ch + 1 тип переменной ch расширяется до int для выполнения вычислений, а результирующее значение int передается в функцию putchar(), которая принимает аргумент типа int, но при отображении символа задействует только последний байт этого значения.

Семейство функций для работы с символами ctype.h

Обратите внимание, что в выводе программы из листинга 7.2 точка была преобразована в косую черту; причина в том, что в ASCII код символа косой черты на единицу больше, чем код символа точки. Однако если цель программы заключается в преобразовании только букв, было бы неплохо оставлять неизменными все небуквенные символы, а не только пробелы. Логические операции, обсуждаемые позже в главе, предоставляют способ проверки, не является ли символ пробелом, запятой и т.д., но перечислять все возможные варианты было бы довольно утомительно. К счастью, в С имеется стандартный набор функций для анализа символов; их прототипы содержатся в заголовочном файле `ctype.h`. Эти функции принимают символ в качестве аргумента и возвращают ненулевое значение (истина), если символ принадлежит к конкретной категории, и ноль (ложь) в противном случае. Например, функция `isalpha()` возвращает ненулевое значение, если ее аргумент является буквой. В листинге 7.3 обобщается программа из листинга 7.2 за счет применения этой функции; здесь также задействована только что рассмотренная укороченная структура цикла.

Листинг 7.3. Программа cypher2.c

```
// cypher2.c -- меняет входные данные, оставляя неизменными символы,
//              которые не являются буквами
#include <stdio.h>
#include <ctype.h>            // для доступа к isalpha()
int main(void)
{
   char ch;
   while ((ch = getchar()) != '\n')
   {
      if (isalpha(ch))         // если это буква,
         putchar(ch + 1);      // отобразить следующую букву
      else                     // в противном случае
         putchar(ch);          // отобразить символ как есть
   }
   putchar(ch);                // вывести символ новой строки
   return 0;
}
```

Ниже показан результат пробного выполнения программы; обратите внимание, что строчные и прописные буквы изменяются, а пробелы и знаки препинания — нет:

```
Look! It's a programmer!
Mppl! Ju't b qsphsbnnfs!
```

В табл. 7.1 и 7.2 перечислены функции, предоставляемые в результате включения заголовочного файла `ctype.h`. Кое-где упоминается локаль; это относится к возможности языка С указывать локаль, которая модифицирует или расширяет базовое использование С. (Например, во многих странах в записи дробных частей применяется запятая, а не десятичная точка, и отдельная локаль могла бы указывать, что при выводе данных с плавающей запятой должна использоваться запятая, приводя к отображению `123.45` как `123,45`.) Обратите внимание, что функции отображения не изменяют исходный аргумент, а вместо этого возвращают модифицированное значение. То есть оператор

```
tolower(ch);      // не приводит к изменению ch
```

не изменяет значение переменной `ch`. Чтобы изменить `ch`, запишите следующий оператор:

```
ch = tolower(ch); // преобразовать ch к нижнему регистру
```

Таблица 7.1. Функции проверки символьных значений в `ctype.h`

Имя функции	Возвращает истинное значение, если аргумент является указанным ниже
`isalnum()`	Алфавитно-цифровой (буквенный или цифровой)
`isalpha()`	Алфавитный
`isblank()`	Стандартный пробельный символ (пробел, горизонтальная табуляция либо новая строка) или любой дополнительный символ подобного рода, специфичный для локали
`iscntrl()`	Управляющий символ, такой как <Ctrl+B>
`isdigit()`	Цифра
`isgraph()`	Любой печатаемый символ, отличный от пробела
`islower()`	Символ нижнего регистра
`isprint()`	Печатаемый символ
`ispunct()`	Символ пунктуации (любой печатаемый символ, отличный от пробела или алфавитно-цифрового символа)
`isspace()`	Пробельный символ (символ пробела, новой строки, перевода страницы, возврата каретки, вертикальной или горизонтальной табуляции или, возможно, другой символ, определенный локалью)
`isupper()`	Символ верхнего регистра
`isxdigit()`	Символ шестнадцатеричной цифры

Таблица 7.2. Функции отображения символов в `ctype.h`

Имя функции	Действие
`tolower()`	Если аргумент является символом верхнего регистра, функция возвращает его версию в нижнем регистре; в противном случае она возвращает исходный аргумент
`toupper()`	Если аргумент является символом нижнего регистра, функция возвращает его версию в верхнем регистре; в противном случае она возвращает исходный аргумент

Множественный выбор `else if`

Жизнь нередко ставит нас перед выбором из более чем двух вариантов. Чтобы учесть этот факт, структуру `if else` можно расширить посредством конструкции `else if`. Давайте рассмотрим конкретный пример. Коммунальные предприятия часто выставляют счета за электроэнергию в зависимости от потребленного объема. Ниже приведены тарифы на потребленную электроэнергию в одной из таких компаний, основанные на киловатт-часах (кВт/ч).

Первые 360 кВт/ч:	$0.13230 за 1 кВт/ч
Следующие 108 кВт/ч	$0.15040 за 1 кВт/ч
Следующие 252 кВт/ч	$0.30025 за 1 кВт/ч
Свыше 720 кВт/ч	$0.34025 за 1 кВт/ч

Если вы намерены вести учет расхода электроэнергии, то имеет смысл написать программу для вычисления стоимости потребленной электроэнергии. Программа в листинге 7.4 является первым шагом в этом направлении.

Листинг 7.4. Программа electric.c

```
// electric.c -- подсчитывает сумму для счета за электроэнергию
#include <stdio.h>
#define RATE1 0.13230           // тариф за первые 360 кВт/ч
#define RATE2 0.15040           // тариф за следующие 108 кВт/ч
#define RATE3 0.30025           // тариф за следующие 252 кВт/ч
#define RATE4 0.34025           // тариф, когда расход превышает 720 кВт/ч
#define BREAK1 360.0            // первая точка разрыва тарифов
#define BREAK2 468.0            // вторая точка разрыва тарифов
#define BREAK3 720.0            // третья точка разрыва тарифов
#define BASE1 (RATE1 * BREAK1)
// стоимость 360 кВт/ч
#define BASE2 (BASE1 + (RATE2 * (BREAK2 - BREAK1)))
// стоимость 468 кВт/ч
#define BASE3 (BASE1 + BASE2 + (RATE3 *(BREAK3 - BREAK2)))
// стоимость 720 кВт/ч
int main(void)
{
   double kwh;                  // израсходованные киловатт-часы
   double bill;                 // сумма к оплате

   printf("Введите объем израсходованной электроэнергии в кВт/ч.\n");
   scanf("%lf", &kwh);          // %lf для типа double
   if (kwh <= BREAK1)
      bill = RATE1 * kwh;
   else if (kwh <= BREAK2)      // количество кВт/ч в промежутке от 360 до 468
      bill = BASE1 + (RATE2 * (kwh - BREAK1));
   else if (kwh <= BREAK3)      // количество кВт/ч в промежутке от 468 до 720
      bill = BASE2 + (RATE3 * (kwh - BREAK2));
   else                         // количество кВт/ч превышает 680
      bill = BASE3 + (RATE4 * (kwh - BREAK3));
   printf("Сумма к оплате за %.1f кВт/ч составляет $%1.2f.\n", kwh, bill);

   return 0;
}
```

Вот пример вывода:

```
Введите объем израсходованной электроэнергии в кВт/ч.
580
Сумма к оплате за 580.0 кВт/ч составляет $97.50.
```

В программе из листинга 7.4 для представления тарифов применяются символические константы, которые для удобства собраны в одном месте. Если компания-производитель электроэнергии меняет свои тарифы (это возможно), наличие их в одном месте упрощает модификацию. В листинге также используются символические константы для точек разрыва. Они тоже могут изменяться. Константы BASE1 и BASE2 выражены через тарифы и точки разрыва. Таким образом, если тарифы и точки разрыва меняются, значения BASE1 и BASE2 обновляются автоматически. Вы можете вспомнить, что препроцессор не выполняет вычислений. Там, где в программе появляется константа BASE1, она заменяется выражением `0.13230 * 360.0`. Компилятор вычислит числовое значение этого выражения (`47.628`) и в окончательном коде программы будет присутствовать число `47.628`, а не выражение.

Поток программы прямолинеен. В зависимости от значения переменной kwh выбирается одна из трех формул. Вы должны уделить особое внимание тому факту, что единственным условием попадания программы на первый else является ввод значения kwh, которое равно или больше 360. Таким образом, строка else if (kwh <= BREAK2) в действительности эквивалентна требованию, чтобы значение kwh находилось в пределах от 360 до 482, как указано в комментариях. Подобным же образом, финальная конструкция else может быть достигнута, только когда значение kwh превышает 720. И, наконец, обратите внимание, что константы BASE1, BASE2 и BASE3 представляют общую стоимость для первых 360, 468 и 720 киловатт-часов соответственно. Поэтому необходимо суммировать только дополнительные затраты за электроэнергию, потребленную сверх указанных объемов.

Фактически конструкция else if — это вариация того, что вы уже знаете. Например, основная часть программы представляет собой всего лишь другой способ написания следующего кода:

```
if (kwh <= BREAK1)
   bill = RATE1 * kwh;
else
   if (kwh <= BREAK2)          // количество кВт/ч в промежутке от 360 до 468
      bill = BASE1 + (RATE2 * (kwh - BREAK1));
   else
      if (kwh <= BREAK3)       // количество кВт/ч в промежутке от 468 до 720
         bill = BASE2 + (RATE3 * (kwh - BREAK2));
      else                     // количество кВт/ч превышает 680
         bill = BASE3 + (RATE4 * (kwh - BREAK3));
```

Программа состоит из оператора if else, в части else которого указан другой оператор if else. Про второй оператор if else говорят, что он *вложен* в первый, а про третий — что он вложен во второй. Вспомните, что вся структура if else считается одним оператором, поэтому мы не обязаны заключать вложенные операторы if else в фигурные скобки. Однако использование скобок прояснило бы назначение этого конкретного формата.

Две показанных формы практически эквивалентны. Единственное различие в том, где размещаются пробелы и новые строки, но компилятор данный факт игнорирует. Тем не менее, первая форма предпочтительнее, поскольку она более ясно демонстрирует выбор из четырех возможностей. Эта форма позволяет легко увидеть существующие варианты выбора даже при беглом взгляде на программу. Применяйте отступы для вложенных операторов там, где они нужны — например, когда вы должны проверять две разных величины. Примером такой ситуации является повышенная на 10% плата за потребление электроэнергии свыше 720 киловатт-часов в летние месяцы.

Вы можете выстраивать в цепочку столько операторов else if, сколько необходимо (разумеется, в рамках ограничений компилятора), как иллюстрируется в следующем фрагменте кода:

```
if (score < 1000)
    bonus = 0;
else if (score < 1500)
    bonus = 1;
else if (score < 2000)
    bonus = 2;
else if (score < 2500)
    bonus = 4;
else
    bonus = 6;
```

(Этот фрагмент может быть частью игровой программы, в которой переменная bonus представляет собой количество дополнительных "питательных таблеток", которые игрок получает за очередной круг.)

Относительно ограничений компилятора следует отметить, что в стандарте C99 от компилятора требуется поддержка не менее 127 уровней вложенности.

Образование пар else и if

Когда в программе присутствует множество конструкций if и else, как компилятор решает, какой if какому else соответствует? В качестве примера рассмотрим следующий фрагмент программы:

```
if (number > 6)
   if (number < 12)
      printf("Вы закончили игру!\n");
else
   printf("К сожалению, вы потеряли право хода!\n");
```

В каком случае выводится сообщение "К сожалению, вы потеряли право хода!" – когда значение переменной number меньше или равно 6 либо когда значение number больше 12? Другими словами, к какому if относится else – к первому или ко второму? Правильный ответ таков: else относится ко второму if. То есть вы получите следующие ответы.

Число	Результат
5	Отсутствует
10	Вывод сообщения "Вы закончили игру!"
15	Вывод сообщения "К сожалению, вы потеряли право хода!"

Правило устанавливает, что else относится к самому последнему if, если только фигурные скобки не указывают другое (рис. 7.2).

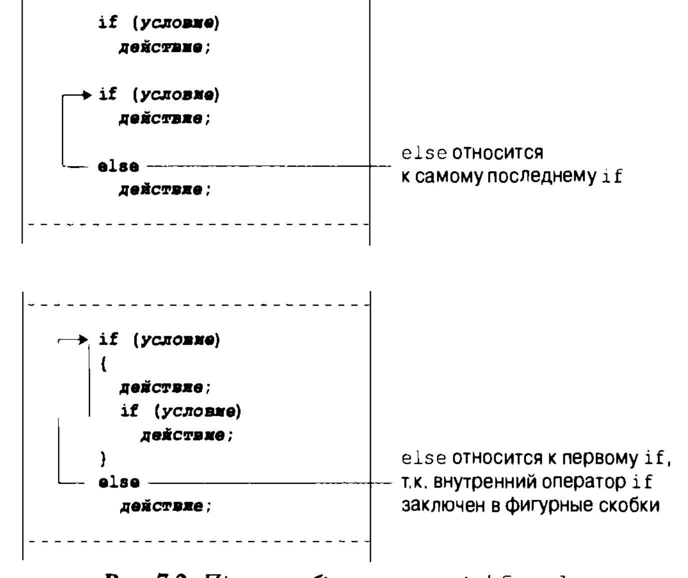


Рис. 7.2. Правило образования пар if и else

На заметку!

Снабдите предпоследнюю конструкцию *действие;* отступом в два пробела и завершите последнюю конструкцию *действие* точкой с запятой. Сместите фигурные скобки } и { влево на две позиции.

Отступы в первом примере расставлены так, чтобы выглядело, будто else относится к первому if, но помните, что компилятор игнорирует отступы. Если действительно необходимо отнести else первому if, фрагмент кода можно было переписать следующим образом:

```
if (number > 6)
{
   if (number < 12)
      printf("Вы закончили игру!\n");
}
else
   printf("К сожалению, вы потеряли право хода!\n");
```

Теперь ответы были бы следующими.

Число	Результат
5	Вывод сообщения "К сожалению, вы потеряли право хода!"
10	Вывод сообщения "Вы закончили игру!"
15	Отсутствует

Другие вложенные операторы `if`

Вы уже видели, что последовательность операторов if...else if...else является формой вложенного оператора if, которая позволяет делать выбор из набора вариантов. Другой вид вложенного оператора if используется, когда выбор конкретного варианта приводит к дополнительному выбору. Например, в программе с помощью оператора if else может осуществляться выбор между мужчинами и женщинами. Каждая ветвь внутри if else, в свою очередь, может содержать еще один оператор if else, предназначенный для различения групп с разным доходом.

Давайте применим эту форму вложенного if при решении следующей задачи. Для заданного целого числа нужно вывести все целые числа, на которые заданное число делится без остатка; если таких делителей нет, необходимо вывести сообщение о том, что число является простым.

Описанная задача требует некоторого обдумывания, прежде чем браться за написание кода. Первым делом, понадобится спроектировать общую структуру программы. Для удобства в программе должен использоваться цикл, позволяющий вводить проверяемые числа. В таком случае не придется запускать программу каждый раз, когда нужно исследовать новое число. Мы уже создали модель для цикла этого вида:

```
вывести пользователю приглашение на ввод числа
пока возвращаемым значением функции scanf() является 1
   проанализировать число и сообщить результаты
   вывести пользователю приглашение на ввод числа
```

Вспомните, что за счет применения функции scanf() в условии проверки цикла программа пытается прочитать число и проверить, должен ли цикл быть продолжен.

Далее потребуется выработать план для поиска делителей. Вероятно, наиболее очевидный подход выглядит примерно так:

```
for (div = 2; div < num; div++)
    if (num % div == 0)
        printf("%d делится на %d\n", num, div);
```

В цикле проверяются все числа в промежутке между 2 и num для выяснения, делится ли значение num на них без остатка. К сожалению, такой подход является затратным в смысле времени. Можно поступить гораздо лучше. Рассмотрим, например, процесс поиска делителей для числа 144. Вы обнаруживаете, что 144 % 2 дает 0, т.е. 144 делится на 2 без остатка. Если затем действительно выполнить деление 144 на 2, получится число 72, которое также является делителем 144, так что результатом успешной проверки num % div будут два делителя, а не один. Однако главное достоинство такого подхода кроется в изменении пределов при проверке условия завершения цикла. Чтобы увидеть, как это работает, взгляните на пары делителей, полученные в процессе выполнения цикла: 2, 72; 3, 48; 4, 36; 6, 24; 8, 18; 9, 16; 12, 12; 16, 9; 18, 8 и т.д. Видите, в чем дело? После пары 12, 12 вы начинаете получать те же самые делители (в обратном порядке), которые уже были найдены. Вместо того чтобы продолжать цикл до 143, вы можете остановиться по достижении 12. Это существенно сокращает количество итераций.

Обобщая это открытие, вы увидите, что должны выполнять проверку только до значения, равного квадратному корню num, а не полному num. Для чисел вроде 9 выигрыш не слишком велик, но для чисел порядка 10 000 и выше он огромен. Однако вместо того чтобы иметь дело с квадратными корнями, условие проверки можно выразить следующим образом:

```
for (div = 2; (div * div) <= num; div++)
    if (num % div == 0)
         printf("%d делится на %d и %d.\n",
                num, div, num / div);
```

Если num имеет значение 144, цикл выполняется до div, равного 12. Если num имеет значение 145, цикл выполняется до div, равного 13.

Есть две причины для использования такой проверки, а не проверки с извлечением квадратного корня. Во-первых, целочисленное умножение выполняется быстрее, чем извлечение квадратного корня. Во-вторых, формально функция вычисления квадратного корня еще не была представлена.

Мы должны решить еще две проблемы, после чего можно приступать к программированию. Во-первых, как быть, если проверяемое число является точным квадратом? Сообщение о том, что 144 делится на 12 и еще раз на 12 выглядит неуклюже, поэтому можно предусмотреть вложенный оператор if, в котором проверять равенство div и num / div и в таком случае выводить один делитель вместо двух.

```
for (div = 2; (div * div) <= num; div++)
{
    if (num % div == 0)
    {
        if (div * div != num)
            printf("%d делится на %d и %d.\n",
                num, div, num / div);
        else
            printf("%d делится на %d.\n", num, div);
    }
}
```

На заметку!

Формально `if else` считается одним оператором, поэтому помещать его в фигурные скобки нет необходимости. Внешний `if` – тоже отдельный оператор, так что скобки для него не нужны. Однако когда операторы становятся длинными, скобки упрощают понимание происходящего, а также служат защитой на тот случай, если вы в дальнейшем добавите еще один оператор в `if` или в цикл.

Во-вторых, как узнать, что число является простым? Если значение `num` является простым, то поток управления программы никогда не попадет внутрь оператора `if`. Для решения этой проблемы можно присвоить некоторой переменной какое-то значение, скажем, `1`, за пределами цикла и установить ее в `0` внутри `if`. После завершения цикла можно проверить переменную на предмет равенства `1`. Если это так, то вход в тело оператора `if` не совершался, и число является простым. Переменную подобного рода часто называют *флагом*.

Традиционно в С для флагов применялся тип `int`, но здесь отлично подойдет новый тип `_Bool`. К тому же, включив в программу заголовочный файл `stdbool.h`, для этого типа вместо ключевого слова `_Bool` можно использовать `bool`, а вместо `1` и `0` – идентификаторы `true` и `false`. Все эти идеи воплощены в листинге 7.5. Чтобы расширить диапазон, в программе применяется тип `long` вместо `int`. (Если ваша система не поддерживает тип `_Bool`, можете использовать для переменной `isPrime` тип `int` и применять `1` и `0` вместо `true` и `false`.)

Листинг 7.5. Программа divisors.c

```
// divisors.c -- вложенные операторы if отображают делители числа
#include <stdio.h>
#include <stdbool.h>
int main(void)
{
   unsigned long num;              // проверяемое число
   unsigned long div;              // потенциальные делители
   bool isPrime;                   // флаг простого числа
   printf("Введите целое число для анализа; ");
   printf("для завершения введите q.\n");
   while (scanf("%lu", &num) == 1)
   {
      for (div = 2, isPrime = true; (div * div) <= num; div++)
      {
         if (num % div == 0)
         {
            if ((div * div) != num)
            printf("%lu делится на %lu и %lu.\n",
                  num, div, num / div);
            else
               printf("%lu делится на %lu.\n",
                     num, div);
            isPrime= false;        // число не является простым
         }
      }
      if (isPrime)
         printf("%lu является простым числом.\n", num);
      printf("Введите следующее число для анализа; ");
      printf("для завершения введите q.\n");
   }
   printf("До свидания.\n");
   return 0;
}
```

Обратите внимание на то, что программа использует операцию запятой в управляющем выражении цикла `for`, чтобы предоставить вам возможность инициализации переменной `isPrime` значением `true` при каждом вводе нового числа.

Ниже показан пример выполнения этой программы:

```
Введите целое число для анализа; для завершения введите q.
123456789
123456789 делится на 3 и 41152263.
123456789 делится на 9 и 13717421.
123456789 делится на 3607 и 34227.
123456789 делится на 3803 и 32463.
123456789 делится на 10821 и 11409.
Введите следующее число для анализа; для завершения введите q.
149
149 является простым числом.
Введите следующее число для анализа; для завершения введите q.
2013
2013 делится на 3 и 671.
2013 делится на 11 и 183.
2013 делится на 33 и 61.
Введите следующее число для анализа; для завершения введите q.
q
До свидания.
```

Программа будет идентифицировать 1 как простое число, хотя формально это не так. Логические операции, обсуждаемые в следующем разделе, позволят исключить 1 из списка простых чисел.

Сводка: использование операторов `if` для реализации выбора

Ключевые слова

`if, else`

Общий комментарий

В каждой из приведенных ниже форм конструкция *оператор* может быть простым или составным оператором. Истинным является выражение, имеющее ненулевое значение.

Форма 1

```
if (выражение)
   оператор
```

оператор выполняется, когда *выражение* принимает истинное значение.

Форма 2

```
if (выражение)
   оператор1
else
   оператор2
```

Если *выражение* истинно, выполняется *оператор1*, в противном случае – *оператор2*.

Форма 3

```
if (выражение1)
   оператор1
else if (выражение2)
   оператор2
else
   оператор3
```

Если *выражение1* истинно, выполняется *оператор1*. Если *выражение1* ложно, но *выражение2* истинно, выполняется *оператор2*. Если оба выражения ложны, выполняется *оператор3*.

Пример

```
if (legs == 4)
   printf("Это может быть лошадь.\n");
else if (legs > 4)
   printf("Это не лошадь.\n");
else /* случай, когда ног меньше 4 */
{
   legs++;
   printf("Теперь на одну ногу больше.\n");
}
```

Давайте будем логичными

Ранее вы видели, что в операторах `if` и `while` в качестве условий проверки часто применяются выражения отношений. Временами возникает необходимость в объединении двух и более выражений. Например, предположим, что требуется программа, которая подсчитывает во введенном предложении количество символов, отличных от одиночных и двойных кавычек. В данном случае можно использовать логические операции и символ точки (`.`) для идентификации конца предложения. В листинге 7.6 представлена короткая программа, иллюстрирующая этот метод.

Листинг 7.6. Программа chcount.c

```
// chcount.c -- использование логической операции "И"
#include <stdio.h>
#define PERIOD '.'
int main(void)
{
   char ch;
   int charcount = 0;

   while ((ch = getchar()) != PERIOD)
   {
      if (ch != '"' && ch != '\'')
         charcount++;
   }
   printf("Здесь имеется %d символов, отличных от кавычек.\n", charcount);

   return 0;
}
```

Вот пример выполнения этой программы:

```
Я не читал бестселлер "Я ничего не смыслю в программировании".
Здесь имеется 60 символов, отличных от кавычек.
```

Действие начинается с чтения символа и проверки, не является ли этот символ точкой, поскольку точка обозначает конец предложения. Далее появляется кое-что новое: в операторе присутствует логическая операция "И" — `&&`. Оператор `if` можно трактовать так: если символ не является двойной кавычкой *И* не является одиночной кавычкой, то увеличить значение `charcount` на 1.

Чтобы все выражение целиком было истинным, должны быть истинными оба условия. Логические операции имеют более низкий приоритет, чем условные операции, так что дополнительные круглые скобки для подвыражений не нужны.

В С доступны три логических операции.

Операция	Описание
&&	"И"
||	"ИЛИ"
!	"НЕ"

Предположим, что exp1 и exp2 — два простых условных выражения, такие как cat > rat и debt == 1000. Тогда справедливы следующие утверждения.

- exp1 && exp2 истинно, только если истинны оба выражения exp1 и exp2.
- exp1 || exp2 истинно, если истинно либо выражение exp1, либо выражение exp2 или оба.
- !exp1 истинно, если exp1 ложно, и ложно, если exp1 истинно.

Рассмотрим несколько конкретных примеров.

- 5 > 2 && 4 > 7 ложно, поскольку истинно только одно из подвыражений.
- 5 > 2 || 4 > 7 истинно, т.к., по меньшей мере, одно из подвыражений истинно.
- !(4 > 7) истинно, потому что 4 не больше 7.

Кстати, последний пример эквивалентен следующему выражению:

```
4 <= 7
```

Если вы не знакомы или недостаточно освоились с логическими операциями, не забывайте о простой истине:

```
(практика && время) == совершенство
```

Альтернативное представление: заголовочный файл iso646.h

Язык С разрабатывался в США на системах, оснащенных стандартными для США клавиатурами. Однако в мире не все клавиатуры имеют те же самые символы, что и клавиатуры в США. В связи с этим стандарт С99 вводит альтернативные формы написания логических операций, которые определены в заголовочном файле iso646.h. Включив этот файл в программу, вы можете указывать and вместо &&, or вместо || и not вместо !. Например, фрагмент

```
if (ch != '"' && ch != '\'')
   charcount++;
```

можно переписать следующим образом:

```
if (ch != '"' and ch != '\'')
   charcount++;
```

Возможные варианты перечислены в табл. 7.3; запомнить их довольно легко. Может возникнуть вопрос, почему в С просто не используются эти новые термины. Вероятнее всего потому, что исторически в языке С пытались обходиться минимальным количеством ключевых слов. В справочном разделе V приложения Б приведен список дополнительных альтернативных форм записи для операций, с которыми вы пока еще не встречались.

Таблица 7.3. Альтернативное представление логических операций

Традиционное представление	Представление посредством `iso646.h`
`&&`	`and`
`||`	`or`
`!`	`not`

Приоритеты операций

Операция ! имеет очень высокий уровень приоритета – больше, чем у операции умножения, и такой же, как у операции инкремента; выше него только приоритет круглых скобок. Операция && имеет более высокий приоритет, чем ||, и обе они по приоритету уступают условным операциям, но превосходят операцию присваивания. Таким образом, выражение

```
a > b && b > c || b > d
```

интерпретируется так, как показано ниже:

```
((a > b) && (b > c)) || (b > d)
```

То есть значение b находится между a и c или значение b больше, чем d.

Многие программисты указывают скобки даже там, где они не обязательны, как во второй версии. Это позволяет понять смысл выражения тем, кто не очень хорошо помнит приоритеты логических операций.

Порядок вычисления выражений

Помимо случаев, когда две операции совместно используют один операнд, в С совершенно не гарантируется, какие части сложного выражения будут вычислены первыми. Например, в следующем операторе подвыражение 5 + 3 может быть вычислено как раньше подвыражения 9 + 6, так и позже:

```
apples = (5 + 3) * (9 + 6);
```

Такая неопределенность оставлена в языке для того, чтобы разработчики компиляторов могли выбрать наиболее эффективный вариант для конкретной системы. Одним исключением из этого правила (или отсутствие правила) является обработка логических операций. В С гарантируется порядок вычисления логических выражений слева направо. Операции && и || – это точки следования, так что все побочные эффекты происходят до перехода с одного операнда к другому. Более того, это гарантирует, что как только обнаруживается элемент, из-за которого все выражение становится ложным, вычисление прекращается. Такой подход делает возможным применение конструкций, подобных показанной ниже:

```
while ((c = getchar()) != ' ' && c != '\n')
```

Здесь устанавливается цикл, который читает символы до появления символа пробела или новой строки. Первое подвыражение присваивает значение переменной c, которая затем используется во втором подвыражении. Не имея этой гарантии порядка, компьютер может попытаться вычислить значение второго подвыражения, прежде чем переменная c получит значение.

Вот еще один пример:

```
if (number != 0 && 12/number == 2)
   printf("Значение переменной number равно 5 или 6.\n");
```

Если переменная number имеет значение 0, то первое подвыражение ложно, и вычисление условного выражения дальше не продолжается. Это уберегает от переполнения, связанного с попыткой деления на ноль. Многие языки не обладают такой особенностью. Увидев, что значение переменной number равно 0, они все равно переходят к проверке следующего условия.

Наконец, рассмотрим такой пример:

```
while (x++ < 10 && x + y < 20)
```

Тот факт, что операция && является точкой следования, служит гарантией того, что значение x будет инкрементировано до вычисления выражения справа.

Сводка: логические операции и выражения

Логические операции

В логических операциях в качестве операндов обычно выступают выражения отношений. Операция ! принимает один операнд. Остальные операции выполняются над двумя операндами, один из которых находится слева от знака операции, другой — справа.

Операция	Описание
&&	"И"
||	"ИЛИ"
!	"НЕ"

Логические выражения

Результат выражение1 && выражение2 истинный тогда и только тогда, когда истинны и выражение1, и выражение2. Результат выражение1 || выражение2 истинный, если истинно любое из выражений или оба. Результат !выражение истинный, если выражение ложно, и наоборот.

Порядок вычисления

Логические выражения вычисляются слева направо. Вычисление прекращается, как только обнаруживается что-то, что делает ложным все выражение.

Примеры

6 > 2 && 3 == 3 — истинно.

! (6 > 2 && 3 == 3) — ложно.

!= 0 && (20 / x) < 5 — второе подвыражение вычисляется, только если значение x является ненулевым.

Диапазон значений

Операция && позволяет проверять вхождение в диапазоны значений. Например, чтобы проверить, находится ли значение переменной score в диапазоне от 90 до 100, можно применить следующий оператор:

```
if (range >= 90 && range <= 100)
   printf("Неплохой результат!\n");
```

Важно избегать имитации общей системы обозначений, принятой в математике, как показано в этом примере:

```
if (90 <= range <= 100)   // Не поступайте так!
   printf("Неплохой результат!\n");
```

Проблема в том, что данный код содержит семантическую, а не синтаксическую ошибку, поэтому компилятор не сможет ее обнаружить (хотя может выдать предупреждающее сообщение). Поскольку для выполнения операции <= принят порядок слева направо, проверочное выражение интерпретируется так:

```
(90 <= range) <= 100
```

Подвыражение 90 <= range получает либо значение 1 (истина), либо 0 (ложь). И то, и другое значение меньше 100, поэтому все выражение всегда истинно вне зависимости от значения range. Таким образом, для проверки на вхождение в диапазон следует прим пользоваться операцией &&.

Во многих программах проверка вхождения в диапазон применяется для определения того, является ли символ, скажем, строчной буквой. Например, предположим, что переменная ch имеет тип char:

```
if (ch >= 'a' && ch <= 'z')
   printf("Это строчная буква.\n");
```

Этот фрагмент работает для таких символьных кодов, как ASCII, в которых коды идущих друг за другом букв являются последовательно возрастающими числами. Тем не менее, он не будет работать с рядом других кодировок, включая EBCDIC. Более переносимый способ реализации такой проверки предполагает использование функции islower() из заголовочного файла ctype.h (см. табл. 7.1):

```
if (islower(ch))
   printf("Это строчный символ.\n");
```

Функция islower() не зависит от применяемой кодировки символов. (Однако в некоторых устаревших реализациях семейство функций ctype.h отсутствует.)

Программа подсчета слов

Теперь вы располагаете всеми инструментами для написания программы подсчета слов (т.е. программы, которая читает входной текст и сообщает количество найденных в нем слов). Параллельно можно также подсчитывать символы и строки. Давайте посмотрим, что такая программа должна включать.

Во-первых, программа должна выполнять посимвольный ввод, а также иметь возможность узнавать, когда останавливаться. Во-вторых, она должна быть способна распознавать и подсчитывать такие элементы, как символы, строки и слова. Вот представление этой программы в виде псевдокода:

```
читать символ
пока еще имеются входные данные
   инкрементировать счетчик символов
   если строка прочитана, инкрементировать счетчик строк
   если слово прочитано, инкрементировать счетчик слов
   читать следующий символ
```

Вы уже имели дело с моделью цикла ввода:

```
while ((ch = getchar()) != STOP)
{
...
}
```

Здесь STOP представляет некоторое значение для ch, сигнализирующее о конце ввода. В примерах, рассмотренных до сих пор, для этой цели использовались символы новой строки и точки, однако ни один из них не подходит для универсальной про-

граммы подсчета слов. На данный момент выберем символ (такой как, например, |), который редко встречается в тексте. В главе 8 будет продемонстрировано более удачное решение, которое также позволит применять программу с текстовыми файлами и клавиатурным вводом.

Теперь приступим к рассмотрению тела цикла. Так как для ввода в программе используется функция getchar(), подсчет символов можно вести, инкрементируя счетчик при каждой итерации цикла. Для подсчета количества строк программа может проверять наличие символов новой строки. Если программа сталкивается с символом новой строки, она должна инкрементировать счетчик строк. Потребуется еще решить, что делать, если символ STOP встречается в середине строки. Должен ли он учитываться как строка? Один из ответов предполагает трактовку строки как неполной, т.е. строки, в которой содержатся различные символы, но нет символа конца строки. Этот случай можно идентифицировать, отслеживая предыдущий прочитанный символ. Если прочитанный символ, предшествующий STOP, не является символом новой строки, то вы имеете неполную строку.

Наиболее запутанная часть программы касается идентификации слов. Прежде всего, необходимо дать определение, что понимается под словом. Давайте выберем относительно простой подход и определим слово как последовательность символов, которая не содержит пробельных символов (символов пробела, табуляции и новой строки). В этом смысле "glymxck" и "r2d2" являются словами. Слово начинается, когда программа впервые встречает символ, отличный от пробельного, и заканчивается, когда появляется следующий пробельный символ. Ниже показано простейшее проверочное выражение, которое обеспечивает обнаружение символов, отличных от пробельных:

```
c != ' ' && c != '\n' && c != '\t'   /* истинно, если с - не пробельный символ */
```

Наиболее прямолинейное проверочное выражение, обеспечивающее выявление пробельных символов, имеет следующий вид:

```
c != ' ' || c == '\n' || c == '\t'  /* истинно, если с - пробельный символ */
```

Однако проще применить функцию isspace() из ctype.h, которая возвращает значение true, если переданный ей аргумент представляет собой пробельный символ. Таким образом, функция isspace(c) возвращает истинное значение, если c – пробельный символ, и !isspace(c) будет истинным, если c таковым не является.

Чтобы отслеживать, входит ли символ в слово, при считывании первого символа слова можно устанавливать в 1 некоторый флаг (назовем его inword). В этой точке можно также инкрементировать счетчик слов. Затем до тех пор, пока значение inword остается равным 1 (или истинным), последующие непробельные символы не помечают начало слова. При появлении следующего пробельного символа флаг понадобится сбросить в 0 (или ложь), после чего программа будет готова к поиску следующего слова. Представим все сказанное в виде псевдокода:

```
если с не является пробельным символом и inword ложно
    установить inword в истину и посчитать слово
if с является пробельным символом и inword истинно
    установить флаг inword в ложь
```

При таком подходе inword устанавливается в 1 (истина) в начале каждого слова и в 0 (ложь) в конце каждого слова. Слова подсчитываются только в момент, когда значение inword меняется с 0 на 1. Если вам доступен тип _Bool, можете включить заголовочный файл stdbool.h и использовать ключевое слово bool для типа флага inword, а также true и false для его значений. В противном случае применяйте тип int и значения 1 и 0.

При работе с булевской переменной в качестве условия проверки обычно используют значение этой переменной. То есть применяйте

```
if (inword)
```

вместо

```
if (inword == true)
```

и

```
if (!inword)
```

вместо

```
if (inword == false)
```

Причина в том, что выражение `inword == true` получает значение `true`, если `inword` равно `true`, и `false`, если `inword` равно `false`, поэтому в качестве условия проверки можно применять просто `inword`. Аналогично, `!inword` имеет то же значение, что и выражение `inword == false` (не истинно – `false`, а не ложно – `true`).

В листинге 7.7 описанные идеи (идентификация строк, неполных строк и слов) реализованы на языке С.

Листинг 7.7. Программа wordcnt.c

```
// wordcnt.c -- производит подсчет символов, слов, строк
#include <stdio.h>
#include <ctype.h>            // для isspace()
#include <stdbool.h>          // для bool, true, false
#define STOP '|'
int main(void)
{
   char c;                    // прочитанный символ
   char prev;                 // предыдущий прочитанный символ
   long n_chars = 0L;         // количество символов
   int n_lines = 0;           // количество строк
   int n_words = 0;           // количество слов
   int p_lines = 0;           // количество неполных строк
   bool inword = false;       // == true если символ с находится внутри слова
   printf("Введите текст для анализа (| для завершения):\n");
   prev = '\n';               // используется для идентификации полных строк
   while ((c = getchar()) != STOP)
   {
      n_chars++;              // считать символы
      if (c == '\n')
         n_lines++;           // считать строки
      if (!isspace(c) && !inword)
      {
         inword = true;       // начало нового слова
         n_words++;           // считать слова
      }
      if (isspace(c) && inword)
         inword = false;      // достигнут конец слова
      prev = c;               // сохранить значение символа
   }
   if (prev != '\n')
      p_lines = 1;
   printf("символов = %ld, слов = %d, строк = %d, ",
        n_chars, n_words, n_lines);
   printf("неполных строк = %d\n", p_lines);
   return 0;
}
```

Ниже показан результат выполнения этой программы:

```
Введите текст для анализа (| для завершения):
Reason is a
powerful servant but
an inadequate master.
|
символов = 55, слов = 9, строк = 3, неполных строк = 0
```

Для трансляции псевдокода в код С используются логические операции. Например

```
если c не является пробельным символом и inword ложно
```

преобразуется в следующий код:

```
if (!isspace(c) && !inword)
```

Еще раз обратите внимание, что `!inword` эквивалентно выражению `inword == false`. Полное проверочное выражение определенно более читабельно, чем индивидуальная проверка для каждого пробельного символа:

```
if (c != ' ' && c != '\n' && c != '\t' && !inword)
```

Обе эти формы означают: "если `c` *не* является пробельным символом, *и* если мы *не* находимся внутри слова". Если оба условия удовлетворены, вы должны быть в начале нового слова, и `n_words` инкрементируется. Если вы находитесь посредине слова, то первое условие выполняется, но `inword` будет равно `true`, и `n_words` не инкрементируется. Когда достигается следующий пробельный символ, `inword` снова устанавливается в `false`. Проверьте, правильно ли работает программа в ситуации, когда между словами находится несколько пробелов. В главе 8 будет показано, как модифицировать программу, чтобы она могла подсчитывать слова в файле.

Условная операция ?:

В языке С предлагается сокращенный способ представления одной из форм оператора `if else` — *условное выражение*, для которого применяется условная операция `?:`. Эта операция состоит из двух частей и работает с тремя операндами. Вспомните, что операции с одним операндом называются *унарными*, а с двумя операндами — *бинарными*. Следуя данной традиции, операции с тремя операндами называют *тернарными*, и условная операция является в С единственной в такой категории. Вот пример, выдающий абсолютное значение числа:

```
x = (y < 0) ? -y: y;
```

Все, что находится между знаком = и точкой с запятой, представляет собой условное выражение. Смысл этого оператора можно выразить так: "если `y` меньше нуля, то `x = -y`, иначе `x = y`". С помощью оператора `if else` это можно выразить следующим образом:

```
if (y < 0)
   x = -y;
else
   x = y;
```

Ниже показана общая форма условного выражения:

```
выражение1 ? выражение2 : выражение3
```

Если *выражение1* имеет истинное (ненулевое) значение, то все условное выражение принимает то же значение, что и *выражение2*. Если *выражение1* имеет ложное (нулевое) значение, то все условное выражение получает то же значение, что и *выражение3*.

Условное выражение можно использовать в ситуации, когда переменной необходимо присвоить одно из двух возможных значений. Типичным примером может служить установка переменной в большее из двух значений:

```
max = (a > b) ? a: b;
```

Здесь переменной max присваивается значение a, если оно больше b, и b в противном случае.

Обычно с помощью оператора if else можно достичь того же самого, что и посредством условной операции. Однако версия с условной операцией короче и в зависимости от компилятора может дать в результате более компактный код.

Давайте для примера рассмотрим программу в листинге 7.8. Эта программа вычисляет, сколько банок краски необходимо для того, чтобы покрасить заданное количество квадратных футов поверхности. Основной алгоритм прост: нужно разделить общее число квадратных футов на количество квадратных футов, которые можно покрасить содержимым одной банки. Тем не менее, предположим, что ответом будет 1,7 банки. В магазине можно купить только полные, а не частично заполненные банки, поэтому придется приобрести две банки. Следовательно, программа должна округлять ответ до следующего целого числа. Для обработки такой ситуации применяется условная операция, и она также используется при выводе слова "банка" или "банки".

Листинг 7.8. Программа paint.c

```
/* paint.c -- использование условной операции */
#include <stdio.h>
#define COVERAGE 350      // число квадратных футов на одну банку краски
int main(void)
{
   int sq_feet;
   int cans;

   printf("Введите количество квадратных футов, которые необходимо покрасить:\n");
   while (scanf("%d", &sq_feet) == 1)
   {
      cans = sq_feet / COVERAGE;
      cans += ((sq_feet % COVERAGE == 0)) ? 0 : 1;
      printf("Для этого потребуется %d %s краски.\n", cans,
             cans == 1 ? "банка" : "банки");
      printf("Введите следующее значение (или q для завершения):\n");
   }

   return 0;
}
```

Ниже показан пример выполнения программы:

```
Введите число квадратных футов, которые необходимо покрасить:
349
Для этого потребуется 1 банка краски.
Введите следующее значение (или q для завершения):
351
Для этого потребуется 2 банки краски.
Введите следующее значение (или q для завершения):
q
```

Поскольку в программе применяется тип int, дробная часть результата от деления усекается, т.е. 351/350 дает 1. Таким образом, количество банок округляется до ближайшего меньшего целого. Если sq_feet % COVERAGE равно 0, то sq_feet делится на

COVERAGE без остатка, поэтому значение cans остается без изменений. В противном случае имеется остаток, и значение cans увеличивается на 1. Это достигается с помощью следующего оператора:

```
cans += ((sq_feet % COVERAGE == 0)) ? 0: 1;
```

Он добавляет к cans значение выражения, указанного справа от знака +=. Выражение справа – это условное выражение, принимающее значение 0 или 1 в зависимости от того, делится ли sq_feet на COVERAGE без остатка.

Последний аргумент функции printf() также является условным выражением:

```
cans == 1 ? "банка" : "банки");
```

Если значение переменной cans равно 1, используется строка "банка", в противном случае – строка "банки". Это демонстрирует возможность применения в условной операции строк в качестве второго и третьего операндов.

Сводка: условная операция

Условная операция

?:

Общий комментарий

Эта операция принимает три операнда, каждый из которых является выражением: Операция имеет следующую форму:

выражение1 ? выражение2 : выражение3

Значение всего выражения равно значению *выражение2*, если *выражение1* истинно, и значению *выражение3* в противном случае.

Примеры

(5 > 3) ? 1 : 2 получает значение 1.

(3 > 5) ? 1 : 2 получает значение 2.

(a > b) ? a : b получает большее значение среди a и b.

Вспомогательные средства для циклов: continue и break

Обычно после входа в тело цикла программа выполняет все находящиеся там операторы, прежде чем проводить очередную проверку условия цикла. Операторы continue и break позволяют пропускать часть цикла и даже прекращать его выполнение в зависимости от результатов проверки, производимой внутри тела цикла.

Оператор continue

Этот оператор может использоваться во всех трех формах циклов. Когда он встречается, он вызывает пропуск оставшейся части итерации и начало новой итерации. Если оператор continue указан внутри вложенной структуры, он воздействует только на самую внутреннюю структуру, содержащую его. Давайте опробуем continue в короткой программе, показанной в листинге 7.9.

Листинг 7.9. Программа skippart.c

```
/* skippart.c -- использование оператора continue для пропуска части цикла */
#include <stdio.h>
```

```
int main(void)
{
   const float MIN = 0.0f;
   const float MAX = 100.0f;
   float score;
   float total = 0.0f;
   int n = 0;
   float min = MAX;
   float max = MIN;
   printf("Введите результат первой игры (или q для завершения): ");
   while (scanf("%f", &score) == 1)
   {
      if (score < MIN || score > MAX)
      {
         printf("%0.1f - недопустимое значение. Повторите попытку: ",
               score);
         continue;  // переход к условию проверки цикла while
      }
      printf("Accepting %0.1f:\n", score);
      min = (score < min)? score: min;
      max = (score > max)? score: max;
      total += score;
      n++;
      printf("Введите результат следующей игры (или q для завершения): ");
   }
   if (n > 0)
   {
      printf("Среднее значение %d результатов равно %0.1f.\n", n, total / n);
      printf("Минимальное = %0.1f, максимальное = %0.1f\n", min, max);
   }
   else
      printf("Не было введено ни одного допустимого результата.\n");
   return 0;
}
```

В листинге 7.9 цикл while читает входные данные до тех пор, пока не будет введено нечисловое значение. Оператор if внутри цикла отсеивает недопустимые значения результатов. Если вы, скажем, вводите число 188, программа сообщает о том, что оно является недопустимым. Затем оператор continue заставляет программу пропустить оставшуюся часть цикла, которая предназначена для обработки допустимого входного значения. Взамен программа начинает новую итерацию, считывая очередное входное значение.

Следует отметить, что избежать применения оператора continue можно двумя путями. Один из них предполагает устранение оператора continue и заключение оставшейся части итерации в блок else:

```
if (score < 0 || score > 100)
   /* оператор printf() */
else
{
   /* операторы */
}
```

В качестве альтернативы можно было бы воспользоваться следующим форматом:

```
if (score >= 0 && score <= 100)
{
   /* операторы */
}
```

Преимущество использования continue в этом случае связано с возможностью устранения одного уровня отступа в главной группе операторов. Краткая форма способствует читабельности в ситуации, когда иначе операторы получаются громоздкими и глубоко вложенными.

Оператор continue может также применяться как заполнитель. Например, в следующем цикле читаются и отбрасываются вводимые символы, включая конец строки:

```
while (getchar() != '\n')
    ;
```

Такой прием удобен, когда программа уже прочитала некоторые символы в строке и ей необходимо пропустить оставшиеся символы до начала следующей строки. Проблема этого кода в том, что одиночный символ точки с запятой трудно заметить. Код станет более читабельным, если воспользоваться оператором continue:

```
while (getchar() != '\n')
    continue;
```

Не применяйте оператор continue, если он вместо упрощения усложняет код. Взгляните на следующий фрагмент:

```
while ((ch = getchar() ) != '\n')
{
    if (ch == '\t')
        continue;
    putchar(ch);
}
```

В этом цикле пропускаются символы табуляции, а сам цикл завершается, только когда встретится символ новой строки. Такой цикл можно выразить более экономно:

```
while ((ch = getchar()) != '\n')
    if (ch != '\t')
        putchar(ch);
```

Часто, как и в данном случае, обращение проверки в if устраняет потребность в continue.

Вы уже видели, что оператор continue приводит к пропуску оставшейся части тела цикла. Где в точности возобновляется выполнение цикла? В случае while и do while следующим действием после continue будет вычисление условия проверки цикла. Рассмотрим для примера следующий цикл:

```
count = 0;
while (count < 10)
{
    ch = getchar();
    if (ch == '\n')
        continue;
    putchar(ch);
    count++;
}
```

Цикл считывает 10 символов (исключая символы новой строки, т.к. оператор count++; пропускается, когда значением ch является символ новой строки) и выводит их на экран кроме символа новой строки. После выполнения оператора continue следующим вычисляется проверочное выражение цикла.

В цикле for следующим действием будет вычисление обновляющего выражения и затем проверочного выражения цикла. Взгляните на показанный ниже пример цикла:

```
for (count = 0; count < 10; count++)
{
   ch = getchar();
   if (ch == '\n')
      continue;
   putchar(ch);
}
```

В этом случае после выполнения оператора continue переменная count сначала инкрементируется, а затем сравнивается со значением 10. Следовательно, данный цикл ведет себя несколько иначе цикла while в рассмотренном выше примере. Как и ранее, отображаются только символы, отличные от символа новой строки. Однако на этот раз при подсчете учитываются символы новой строки, так что цикл читает ровно 10 символов, включая символы новой строки.

Оператор break

Оператор break в цикле заставляет программу прервать цикл и перейти к выполнению следующего оператора. В листинге 7.9 замена continue оператором break приводит к выходу из цикла при вводе, скажем, числа 188, а не к пропуску оставшихся операторов внутри цикла и переходу к следующей итерации. На рис. 7.3 приведено сравнение операторов break и continue. Если оператор break находится внутри вложенных циклов, его действие распространяется только на самый внутренний цикл, в котором он содержится.

```
while ( (ch = getchar() ) !=EOF)
{
   blahblah(ch);
   if (ch == '\n')
      break;
   yakyak(ch);
}
blunder(n,m);
```

```
while ( (ch = getchar() ) !=EOF)
{
   blahblah(ch);
   if (ch == '\n')
      continue;
   yakyak(ch);
}
blunder(n,m);
```

Рис. 7.3. Сравнение операторов break *и* continue

Иногда оператор break используется для выхода из цикла, когда для этого существуют две отдельные причины. Программа, показанная в листинге 7.10, содержит цикл для вычисления площади прямоугольника. Цикл прекращается при вводе нечислового значения для длины или ширины прямоугольника.

Листинг 7.10. Программа break.c

```
/* break.c -- использование оператора break для выхода из цикла */
#include <stdio.h>
int main(void)
{
   float length, width;

   printf("Введите длину прямоугольника:\n");
   while (scanf("%f", &length) == 1)
   {
      printf("Длина = %0.2f:\n", length);
      printf("Введите ширину прямоугольника:\n");
      if (scanf("%f", &width) != 1)
         break;
      printf("Ширина = %0.2f:\n", width);
      printf("Площадь = %0.2f:\n", length * width);
      printf("Введите длину прямоугольника:\n");
   }
   printf("Программа завершена.\n");

   return 0;
}
```

Цикл можно было бы реализовать следующим образом:

```
while (scanf("%f %f", &length, &width) == 2)
```

Однако применение оператора break существенно упрощает эхо-вывод вводимых значений.

Как и continue, не используйте оператор break, если это приводит к усложнению кода. Взгляните, например, на следующий цикл:

```
while ((ch = getchar()) != '\n')
{
   if (ch == '\t')
      break;
   putchar(ch);
}
```

Логика программы станет яснее, если выполнять обе проверки в одном месте:

```
while ((ch = getchar() ) != '\n' && ch != '\t')
     putchar(ch);
```

Оператор break является важным дополнением оператора switch, который будет рассматриваться следующим.

Оператор break передает управление оператору, который находится непосредственно после цикла; в отличие от случая с continue внутри цикла for, пропускается обновляющая часть раздела управления цикла. Оператор break, помещенный внутрь вложенного цикла, приводит к выходу только из внутреннего цикла; для выхода из внешнего цикла необходим еще один break:

```
int p, q;
scanf("%d", &p);
while ( p > 0)
{
   printf("%d\n", p);
   scanf("%d", &q);
   while( q > 0)
   {
      printf("%d\n",p*q);
      if (q > 100)
         break;          // выход из внутреннего цикла
      scanf("%d", &q);
   }
   if (q > 100)
      break;             // выход из внешнего цикла
   scanf("%d", &p);
}
```

Выбор из множества вариантов: операторы switch и break

Условная операция и конструкция if else облегчают написание программ, в которых производится выбор между двумя альтернативами. Однако временами в программе должен делаться выбор одного варианта из множества альтернатив. Это можно реализовать с помощью конструкции if else if...else, но во многих случаях удобнее применять оператор switch. Работа этого оператора демонстрируется в листинге 7.11. Данная программа читает букву и отвечает выводом названия животного, которое начинается с такой буквы.

Листинг 7.11. Программа animals.c

```
/* animals.c -- использование оператора switch */
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   char ch;

   printf("Дайте мне букву алфавита, и я укажу вам ");
   printf("название животного,\nначинающееся с этой буквы.\n");
   printf("Введите букву или # для завершения.\n");
   while ((ch = getchar()) != '#')
   {
      if('\n' == ch)
         continue;
      if (islower(ch)) /* только строчные буквы */
         switch (ch)
         {
         case 'а' :
            printf("архар, дикий горный азиатский баран\n");
            break;
         case 'б' :
            printf("бабирусса, дикая малайская свинья\n");
            break;
         case 'к' :
            printf("коати, носуха обыкновенная\n");
            break;
```

```
            case 'в' :
               printf("выхухоль, водоплавающее существо\n");
               break;
            case 'е' :
               printf("ехидна, игольчатый муравьед\n");
               break;
            case 'р' :
               printf("рыболов, светло-коричневая куница\n");
               break;
               default :
               printf("Вопрос озадачил!\n");
            }          /* конец оператора выбора */
         else
            printf("Распознаются только строчные буквы.\n");
         while (getchar() != '\n')
            continue;  /* пропустить оставшуюся часть входной строки */
         printf("Введите следующую букву или # для завершения.\n");
      }               /* конец цикла while */
      printf("До свидания.\n");

      return 0;
}
```

Мы ограничились лишь несколькими буквами, но можно было продолжить в том же духе. Давайте рассмотрим пример выполнения этой программы, после чего проанализируем ее структуру:

```
Дайте мне букву алфавита, и я укажу вам
название животного, начинающееся с этой буквы.
Введите букву или # для завершения.
а [enter]
архар, дикий горный азиатский баран
Введите следующую букву или # для завершения.
вал [enter]
выхухоль, водоплавающее существо
Введите следующую букву или # для завершения.
ф [enter]
Вопрос озадачил!
Введите следующую букву или # для завершения.
Е [enter]
Распознаются только строчные буквы.
Введите следующую букву или # для завершения.
# [enter]
До свидания.
```

Две основных особенности программы касаются использования оператора switch и обработки вводимых данных. Для начала необходимо ознакомиться с тем, как работает оператор switch.

Использование оператора switch

Первым делом вычисляется выражение в круглых скобках, следующее за словом switch. В этом случае оно представляет собой значение, которое переменная ch получила в результате последнего ввода. Затем программа просматривает список *меток* (здесь это case 'а':, case 'б': и т.д.), пока не найдет совпадающее значение. После этого программа переходит на данную строку. А что произойдет, если совпадений не найдено? Если в операторе предусмотрена строка, помеченная как default:, то про-

грамма перейдет на нее. В противном случае выполнение продолжится с оператора, следующего после switch.

Что можно сказать об операторе break? Он заставляет программу выйти из оператора switch и перейти к оператору, находящемуся после switch (рис. 7.4).

```
switch(число)
{
case 1: оператор 1;
        break;
case 2: оператор 2;
        break;
case 3: оператор 3;
        break
default: оператор 4;
}
statement 5;
```

```
switch(число)
{
case 1: оператор 1;
case 2: оператор 2;
case 3: оператор 3;
default: оператор 4;
}
оператор 5;
```

Рис. 7.4. Поток управления в операторах switch *с и без операторов* break

Без break выполнялись бы все операторы между тем, который имеет совпадающую метку, и концом switch. Например, если вы удалите все операторы break, а затем запустите программу и введете букву *в*, будет получен следующий вывод:

```
Дайте мне букву алфавита, и я укажу вам
название животного, начинающееся с этой буквы.
Введите букву или # для завершения.
в [enter]
выхухоль, водоплавающее существо
ехидна, игольчатый муравьед
рыболов, светло-коричневая куница
Вопрос озадачил!
Введите следующую букву или # для завершения.
# [enter]
Программа завершена.
```

Были выполнены все операторы, начиная с case 'в': и до конца оператора switch.

Кстати, break работает и с циклами и с оператором switch, в то время как continue — только с циклами. С другой стороны, continue может применяться внут-

ри оператора switch, если switch находится в цикле. В такой ситуации, как и с другими циклами, continue заставляет программу пропустить остальные операторы цикла, включая другие части switch.

Если вы знакомы с языком Pascal, то можете отметить большое сходство switch с оператором case в Pascal. Самое важное различие между ними связано с тем, что switch требует использования оператора break, если нужно, чтобы выполнялся только помеченный оператор. Кроме того, в конструкции case языка С нельзя применять диапазон.

Проверочное выражение в круглых скобках внутри switch должно иметь целочисленное значение (включая тип char). Метки case должны быть константами целочисленного типа (в том числе char) или целочисленными константными выражениями (выражениями, которые содержат только целочисленные константы). Указывать для метки case переменную не допускается. Ниже приведена структура оператора switch.

```
switch (целочисленное-выражение)
{
   case константа1:
         операторы      ← не обязательно
   case константа2:
         операторы      ← не обязательно
   default:             ← не обязательно
         операторы      ← не обязательно
}
```

Чтение только первого символа строки

Еще одна новая особенность программы animals.c связана с тем, как она читает входные данные. Возможно, вы уже заметили во время выполнения этой программы, что при вводе вап был обработан только первый символ. Такое поведение с отбрасыванием оставшейся части строки часто является желательным в интерактивных программах, ожидающих односимвольные ответы. Указанное поведение обеспечивает следующий код:

```
while (getchar() != '\n')
   continue;        /* пропустить оставшуюся часть входной строки */
```

Этот цикл читает символы из входной последовательности вплоть до символа новой строки, генерируемого нажатием клавиши <Enter>, включительно. Обратите внимание, что возвращаемое значение getchar() не присваивается переменной ch, поэтому программа просто читает символы и отбрасывает их. Так как последним отброшенным будет символ новой строки, то очередной подлежащий чтению символ будет первым символом следующей строки. Функция getchar() считывает его и присваивает переменной ch во внешнем цикле while.

Предположим, что пользователь начинает с нажатия <Enter>, так что первым прочитанным оказывается символ новой строки. Следующий код учитывает эту возможность:

```
if (ch == '\n')
   continue;
```

Множество меток

В операторе switch можно использовать множество меток case, как показано в листинге 7.12.

Листинг 7.12. Программа vowels.c

```
// vowels.c -- использование множества меток
#include <stdio.h>
int main(void)
{
   char ch;
   int a_ct, e_ct, i_ct, o_ct, u_ct;

   a_ct = e_ct = i_ct = o_ct = u_ct = 0;

   printf("Введите текст или # для завершения программы.\n");
   while ((ch = getchar()) != '#')
   {
      switch (ch)
      {
         case 'a' :
         case 'A' : a_ct++;
                    break;
         case 'e' :
         case 'E' : e_ct++;
                    break;
         case 'i' :
         case 'I' : i_ct++;
                    break;
         case 'o' :
         case 'O' : o_ct++;
                    break;
         case 'u' :
         case 'U' : u_ct++;
                    break;
         default  : break;
      }                 // конец оператора switch
   }                    // конец цикла while
   printf("Количество гласных:  A    E    I    O    U\n");
   printf("                     %4d %4d %4d %4d %4d\n",
         a_ct, e_ct, i_ct, o_ct, u_ct);

         return 0;
}
```

Если ch содержит, скажем, букву i, оператор switch переходит в место, помеченное как case 'i':. Поскольку с этой меткой не связан оператор break, управление переходит к следующему оператору, которым является i_ct++;. Если значение ch равно I, управление переходит прямо на этот оператор. В сущности, обе метки ссылаются на один и тот же оператор.

Строго говоря, оператор break для case 'U' не нужен, т.к. благодаря его отсутствию управление в программе перемещается на следующий оператор внутри switch, которым является break для случая default. Следовательно, break для case 'U' можно было убрать, тем самым сократив код. С другой стороны, если позже будут добавляться другие случаи, то наличие оператора break там, где он должен быть, позволит не забыть о необходимости его добавления.

Ниже показан пример выполнения программы:

```
Введите текст или # для завершения программы.
I see under the overseer.#
Количество гласных:  A    E    I    O    U
                     0    7    1    1    1
```

В рассматриваемом случае можно избежать множества меток за счет применения функции toupper() из семейства ctype.h (см. табл. 7.2), преобразовав перед проверкой условия все строчные буквы в прописные:

```
while ((ch = getchar()) != '#')
{
   ch = toupper(ch);
   switch (ch)
   {
      case 'A' : a_ct++;
                 break;
      case 'E' : e_ct++;
                 break;
      case 'I' : i_ct++;
                 break;
      case 'O' : o_ct++;
                 break;
      case 'U' : u_ct++;
                 break;
      default :  break;
   }                          // конец оператора switch
}                             // конец цикла while
```

Либо при желании значение ch можно было бы оставить неизменным и использовать toupper(ch) в качестве условия проверки:

Сводка: выбор из множества вариантов с помощью оператора switch

Ключевое слово

switch

Общий комментарий

Поток управления программы переходит к метке case, содержащей значение конструкции *выражение*. Затем программа выполняет все оставшиеся операторы, если только управление снова не будет перенаправлено посредством оператора break. Как *выражение*, так и метки case должны иметь целочисленные значения (включая тип char), а метки должны быть константами или выражениями, состоящими только из констант. Если ни одна из меток case не совпадает со значением выражения, управление передается оператору, помеченному как default, если таковой предусмотрен. В противном случае управление переходит к оператору, следующему за switch.

Форма

```
switch(toupper(ch))
switch (выражение)
{
   case метка1 : оператор1     // используйте break, чтобы пропустить все
                               // операторы до конца switch
   case метка2 : оператор2
   default     : оператор3
}
```

Помеченных операторов может быть более двух, а случай default является необязательным.

Пример

```
switch (choice)
{
   case 1  :
```

```
    case 2  : printf("Великолепно!\n"); break;
    case 3  : printf("Совершенно верно!\n");
    case 4  : printf("На все сто!\n"); break;
    default : printf("Удачного дня.\n");
}
```

Если переменная choice имеет целое значение 1 или 2, выводится первое сообщение, если 3 — второе и третье сообщения. (Управление переходит на следующий оператор, поскольку после case 3 не указан оператор break.) Если значение choice равно 4, то выводится третье сообщение. Другие значения приводят к выводу только последнего сообщения.

Операторы switch и if else

Когда должен применяться оператор switch, а когда конструкция if else? Часто выбор попросту отсутствует. Оператор switch нельзя использовать, если выбор основан на значении переменной или выражения с плавающей запятой. Точно так же применять оператор switch затруднительно, если переменная должна входить в определенный диапазон. Написать следующий код просто:

```
if (integer < 1000 && integer > 2)
```

К сожалению, охват этого диапазона посредством оператора switch предполагает установку меток case для каждого целого числа в промежутке от 3 до 999. В то же время, когда есть возможность использовать switch, программа зачастую выполняется немного быстрее и имеет более компактный вид.

Оператор goto

Оператор goto, своего рода оплот ранних версий языков BASIC и FORTRAN, доступен и в C. Тем не менее, в отличие от BASIC и FORTRAN, язык C вполне может обойтись без этого оператора. Керниган и Ритчи отзываются об операторе goto как о потенциальном источнике ошибок и советуют применять его как можно реже или вообще не применять. Сначала мы объясним, как использовать goto, а затем покажем, что обычно в нем нет нужды.

Оператор goto состоит из двух частей — ключевое слово goto и имя метки. Именование меток производится по тому же соглашению, которое применяется для имен переменных, например:

```
goto part2;
```

Для корректной работы этого оператора в программе должен присутствовать другой оператор, снабженный меткой part2. Это делается путем помещения перед оператором имени метки с двоеточием:

```
part2: printf("Уточненный анализ:\n");
```

Избегайте goto

В принципе в программе C никогда не возникает потребность в goto, но если ваш опыт программирования основан на языке FORTRAN или BASIC, где этот оператор обязателен, то вполне возможно, что использование goto вошло в привычку. Чтобы помочь вам избавиться от этой зависимости, мы рассмотрим несколько знакомых случаев применения goto и продемонстрируем подход к их решению, характерный для языка C.

- Поддержка ситуации с оператором `if`, требующей выполнения более одного оператора:

```
if (size > 12)
   goto a;
goto b;
a: cost = cost * 1.05;
flag = 2;
b: bill = cost * flag;
```

В ранних версиях BASIC и FORTRAN к оператору `if` относился только один оператор, непосредственно следующий за условием. Не было никаких средств для реализации блоков или составных операторов. Ниже приведен эквивалент этой структуры на языке C. Стандартный для C подход с использованием составного оператора или блока существенно облегчает понимание программы:

```
if (size > 12)
{
   cost = cost * 1.05;
   flag = 2;
}
bill = cost * flag;
```

- Выбор из двух альтернатив:

```
if (ibex > 14)
   goto a;
sheds = 2;
goto b;
a: sheds = 3;
b: help = 2 * sheds;
```

Наличие в C структуры `if else` позволяет выразить выбор более ясно:

```
if (ibex > 14)
   sheds = 3;
else
   sheds = 2;
help = 2 * sheds;
```

В действительности синтаксис более поздних версий BASIC и FORTRAN включает конструкцию `else`.

- Организация бесконечного цикла:

```
readin: scanf("%d", &score);
if (score < 0)
   goto stage2;
множество операторов
goto readin;
stage2: дополнительная обработка;
```

Взамен применяйте цикл `while`:

```
scanf("%d", &score);
while (score <= 0)
{
   множество операторов
   scanf("%d", &score);
}
дополнительная обработка;
```

- Пропуск операторов до конца цикла и начало следующей итерации. Используйте оператор `continue` вместо `goto`.
- Выход из цикла. Применяйте оператор `break` вместо `goto`. Операторы `break` и `continue` фактически являются специализированными формами `goto`. Преимущество их использования заключается в том, что имена операторов позволяют понять их назначение, а также в том, что в них не применяются метки, поэтому отсутствует опасность помещения метки в неподходящее место.
- Беспорядочные переходы между разными частями программы. Избегайте их!

Существует ситуация, в которой многие программисты допускают использование оператора `goto` — выход из набора глубоко вложенных циклов в случае возникновения ошибки (одиночный оператор `break` обеспечит выход только из самого внутреннего цикла):

```
while (funct > 0)
{
   for (i = 1, i <= 100; i++)
   {
      for (j = 1; j <= 50; j++)
      {
         множество операторов;
         if (признак ошибки)
            goto help;
         операторы;
      }
      еще множество операторов;
   }
   другое множество операторов;
}
третье множество операторов;
help: код устранения ошибки;
```

Как можно было заметить в других примерах, альтернативные формы кода яснее для восприятия, чем формы с оператором `goto`. Различия становятся еще более заметными при смешивании нескольких случаев подобного рода. Какие операторы `goto` содействуют операторам `if`, какие эмулируют конструкции `if else`, какие управляют циклами, а какие применены лишь потому, что вы загнали себя в угол? Чрезмерно используя `goto`, вы создаете лабиринт в потоке управления программы. Если вы не знакомы с оператором `goto`, то и не меняйте это положение дел. Если вы привыкли к нему, то постарайтесь отвыкнуть. По иронии судьбы, язык C, который совершенно не нуждается в `goto`, располагает лучшим синтаксисом этого оператора, чем большинство других языков, т.к. позволяет применять для меток описательные слова, а не числа.

Сводка: переходы в программах

Ключевые слова

`break`, `continue`, `goto`

Общий комментарий

Эти операторы заставляют поток управления программы перейти из одного места в другое.

Оператор `break`

Оператор `break` может использоваться с любой из трех форм цикла и оператором `switch`. Он приводит к пропуску оставшихся операторов в теле цикла или внутри `switch` и передаче управления оператору, следующему за оператором цикла или `switch`.

Пример

```
switch (число)
{
   case 4:  printf("Это лучший выбор.\n");
            break;
   case 5:  printf("Это хороший выбор.\n");
            break;
   default: printf("Это плохой выбор.\n");
}
```

Оператор continue

Оператор `continue` может применяться с любой из трех форм циклов, но не с оператором `switch`. Он приводит к пропуску оставшихся операторов цикла. В цикле `while` или `for` начинается новая итерация. В цикле `do while` проверяется условие завершения, после чего при необходимости начинается следующая итерация цикла.

Пример

```
while ((ch = getchar()) != '\n')
{
   if (ch == ' ')
      continue;
   putchar(ch);
   chcount++;
}
```

Этот фрагмент кода выводит и подсчитывает символы, отличные от пробела.

Оператор goto

Оператор `goto` приводит к передаче управления оператору, снабженному указанной меткой. Для отделения помеченного оператора от его метки используется двоеточие. Метки именуются согласно тем же правилам, что и переменные. Помеченный оператор может находиться либо до, либо после оператора `goto`.

Форма

```
goto метка;
   .
   .
   .
метка: оператор
```

Пример

```
top: ch = getchar();
   .
   .
   .
if (ch != 'y')
   goto top;
```

Ключевые понятия

Одним из аспектов интеллекта является возможность подгонять ответы под конкретные обстоятельства. Таким образом, операторы выбора представляют собой основу для разработки программ с интеллектуальным поведением. В языке С выбор реализуется с помощью операторов `if`, `if else` и `switch`, а также условной операции (`?:`).

Для определения того, какие операторы должны быть выполнены, в `if` и `if else` применяется условие проверки. Любое ненулевое значение трактуется как истинное, а ноль — как ложное. Обычно в проверках используются условные выражения, в которых сравниваются два значения, и логические выражения, в которых посредством логических операций создаются сложные выражения.

Следует запомнить один общий принцип: если требуется проверить два условия, нужно применять логическую операцию с двумя завершенными проверочными выражениями. Например, следующий код ошибочен:

```
if (a < x < z)         // неправильно - отсутствует логическая операция
...
if (ch != 'q' && != 'Q') // неправильно - отсутствует завершенное выражение проверки
...
```

Правильный способ предусматривает объединение двух условных выражений с помощью логической операции:

```
if (a < x && x < z)         // использование && для объединения двух выражений
...
if (ch != 'q' && ch != 'Q') // использование && для объединения двух выражений
...
```

Управляющие операторы, представленные в последних двух главах, позволяют разрабатывать намного более мощные и сложные программы, нежели те, с которыми вы имели дело ранее. Чтобы удостовериться в этом, достаточно сравнить некоторые примеры из этой главы с примерами, рассмотренными в предшествующих главах.

Резюме

В данной главе было рассмотрено довольно много тем, так что давайте кратко подытожим. В операторе `if` используется условие проверки для определения того, должен ли быть выполнен одиночный оператор или блок, следующий после условия. Выполнение происходит в случае, если проверочное выражение имеет ненулевое значение, и не происходит, если его значение равно нулю. Оператор `if else` позволяет производить выбор из двух альтернатив. Если условие проверки дает ненулевое значение, выполняется оператор, предшествующий `else`. Если условие проверки имеет нулевое значение, выполняется оператор, следующий за `else`. Указывая непосредственно за `else` еще один оператор `if`, можно построить структуру, которая производит выбор из последовательности альтернатив.

Выражение проверки часто является *выражением отношения*, т.е. выражением, построенным с применением одной из условных операций, таких как < или ==. С помощью логических операций С выражения отношений можно объединять, создавая более сложные проверки.

Условная операция (`?:`) позволяет создавать выражение, которое во многих случаях оказывается более компактной альтернативой оператору `if else`.

Семейство функций для работы с символами `ctype.h`, подобных `isspace()` и `isalpha()`, предлагает удобные инструменты для построения выражений проверки, основанных на классификации символов.

Оператор `switch` позволяет делать выбор из последовательности операторов, помеченных целочисленными значениями. Если целочисленное значение условия проверки, следующего за ключевым словом `switch`, совпадает с какой-то меткой, управление передается оператору, снабженному этой меткой. После этого управление проходит через операторы, следующие за помеченным оператором, до тех пор, пока не встретится оператор `break`.

Наконец, операторы break, continue и goto — это операторы переходов, которые приводят к передаче управления в другое место программы. Оператор break вынуждает программу перейти к оператору, следующему за концом цикла или за оператором switch, который содержит break. Оператор continue заставляет программу пропустить операторы, оставшиеся в теле цикла, и начать новую итерацию.

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля приведены в приложении А.

1. Определите, какие выражения равны true, а какие — false.

   а. `100 > 3 && 'a'>'c'`

   б. `100 > 3 || 'a'>'c'`

   в. `!(100>3)`

2. Напишите выражения для представления следующих условий.

   а. Значение number равно или больше 90, но меньше 100.

   б. Значение ch не является символом q или k.

   в. Значение number находится между 1 и 9 (включая граничные значения), но не равно 5.

   г. Значение number не находится между 1 и 9.

3. В приведенной ниже программе присутствуют излишне сложные выражения отношений, а также откровенные ошибки. Упростите программу и исправьте ошибки.

```
#include <stdio.h>
int main(void)                                          /* 1 */
{                                                       /* 2 */
  int weight, height;       /* вес в фунтах, рост в дюймах */
                                                        /* 4 */
    scanf("%d , weight, height);                        /* 5 */
    if (weight < 100 && height > 64)                    /* 6 */
       if (height >= 72)                                /* 7 */
          printf("Ваш вес слишком мал для вашего роста.\n");
    else if (height < 72 && > 64)                       /* 9 */
       printf("Ваш вес мал для вашего роста.\n");       /* 10 */
  else if (weight > 300 && ! (weight <= 300)            /* 11 */
          && height < 48)                               /* 12 */
    if (!(height >= 48) )                               /* 13 */
      printf("Ваш рост мал для вашего веса.\n");
  else                                                  /* 15 */
    printf("У вас идеальный вес.\n");                   /* 16 */
                                                        /* 17 */
  return 0;
}
```

4. Каковы числовые значения каждого из следующих выражений?

   а. `5 > 2`

   б. `3 + 4 > 2 && 3 < 2`

   в. `x >= y || y > x`

   г. `d = 5 + ( 6 > 2 )`

   д. `'X' > 'T' ? 10: 5`

   е. `x > y ? y > x: x > y`

5. Что выведет следующая программа?

```
#include <stdio.h>
int main(void)
{
   int num;
   for (num = 1; num <= 11; num++)
   {
      if (num % 3 == 0)
         putchar('$');
      else
         putchar('*');
         putchar('#');
      putchar('%');
   }
   putchar('\n');
   return 0;
}
```

6. Что выведет следующая программа?

```
#include <stdio.h>
int main(void)
{
   int i = 0;
   while ( i < 3) {
      switch(i++) {
         case 0 : printf("fat ");
         case 1 : printf("hat ");
         case 2 : printf("cat ");
         default: printf("Oh no!");
      }
      putchar('\n');
   }
   return 0;
}
```

7. Что неправильно в следующей программе?

```
#include <stdio.h>
int main(void)
{
   char ch;
   int lc = 0;   /* счетчик строчных символов
   int lc = 0;   /* счетчик прописных символов
   int lc = 0;   /* счетчик других символов
   while ((ch = getchar()) != '#')
   {
      if ('a' <= ch >= 'z')
         lc++;
      else if (!(ch < 'A') || !(ch > 'Z')
         uc++;
      oc++;
   }
   printf(%d строчных, %d прописных, %d других, lc, uc, oc);
   return 0;
}
```

8. Что выведет следующая программа?

```
/* retire.c */
#include <stdio.h>
int main(void)
{
   int age = 20;
   while (age++ <= 65)
   {
      if (( age % 20) == 0) /* делится ли возраст на 20? */
         printf("Вам %d. Вас повысили в должности.\n", age);
      if (age = 65)
         printf("Вам %d. Получите свои золотые часы.\n", age);
   }
   return 0;
}
```

9. Что выведет следующая программа при указанном вводе?

```
q
c
h
b

#include <stdio.h>
int main(void)
{
   char ch;
   while ((ch = getchar()) != '#')
   {
      if (ch == '\n')
         continue;
      printf("Шаг 1\n");
      if (ch == 'c')
         continue;
      else if (ch == 'b')
         break;
      else if (ch == 'h')
         goto laststep;
      printf("Шаг 2\n");
   laststep: printf("Шаг 3\n");
   }
   printf("Готово\n");
   return 0;
}
```

10. Перепишите программу из вопроса 9 так, чтобы она сохранила свое поведение, но в ней не использовались операторы `continue` и `goto`.

Упражнения по программированию

1. Напишите программу, которая читает входные данные до тех пор, пока не встретится символ #, а затем отображает количество прочитанных пробелов, количество символов новой строки и количество всех остальных символов.

2. Напишите программу, которая читает входные данные до тех пор, пока не встретится символ #. Программа должна выводить каждый введенный символ и его десятичный код ASCII. Каждая строка вывода должна содержать восемь пар "символ-код".

Подсказка: используйте счетчик символов и операцию деления по модулю (%) для вывода символа новой строки на каждой восьмой итерации цикла.

3. Напишите программу, которая читает целые числа до тех пор, пока не встретится число 0. После прекращения ввода программа должна сообщить общее количество введенных четных целых чисел (за исключением 0), среднее значение введенных четных целых чисел, общее количество введенных нечетных целых чисел и среднее значение нечетных чисел.
4. Используя операторы `if else`, напишите программу, которая читает входные данные, пока не встретит символ `#`, заменяет каждую точку восклицательным знаком, изначально присутствующие восклицательные знаки – двумя восклицательными знаками и в конце сообщает о количестве произведенных замен.
5. Выполните упражнение 4, но с применением оператора `switch`.
6. Напишите программу, которая читает входные данные, пока не встретит символ `#`, и сообщает количество вхождений последовательности `ei`.

На заметку!

Эта программа должна "запоминать" предыдущий символ, а также текущий символ. Проверьте ее на входной последовательности вроде "Receive your eieio award#".

7. Напишите программу, которая запрашивает количество часов, отработанных за неделю, и выводит значения общей суммы начислений, налогов и чистой зарплаты. Исходите из перечисленных ниже утверждений.
   а. Основная тарифная ставка заработной платы = $10,00/час
   б. Сверхурочные часы (превышающие 40 часов в неделю) = коэффициент 1,5
   в. Налоговая ставка: 15% с первых $300;
   20% со следующих $150;
   25% с остатка.

   Используйте константы `#define` и не беспокойтесь, если приведенный пример не соответствует действующему налогообложению.
8. Измените предположение а) в упражнении 7 так, чтобы программа предоставляла меню с тарифными ставками. Для выбора тарифной ставки используйте оператор `switch`. После запуска программы вывод должен быть подобным показанному ниже:

```
*****************************************************************
Введите число, соответствующее желаемой тарифной ставке или действию:
1) $8.75/ч                    2) $9.33/ч
3) $10.00/ч                   4) $11.20/ч
5) Выход
*****************************************************************
```

   Если выбран вариант с 1 по 4, программа должна запрашивать количество отработанных часов. Программа должна повторяться до тех пор, пока не будет выбран вариант 5. При вводе чего-то отличного от вариантов 1–5 программа должна напомнить пользователю допустимые варианты для ввода и снова ожидать ввод. Для различных тарифных и налоговых ставок применяйте константы `#define`.
9. Напишите программу, которая принимает в качестве ввода положительное целое число и отображает все простые числа, которые меньше или равны введенному числу.

10. В 1988 году шкала федеральных налоговых ставок Соединенных Штатов была самой простой за все прошедшее время. Она содержала четыре категории, каждая из которых включала две ставки. Ниже приведены самые общие данные (суммы в долларах представляют собой доход, облагаемый налогом).

Категория	Налог
Одинокий	15% с первых $17 850 плюс 28% от суммы, превышающей указанную
Глава семейства	15% с первых $23 900 плюс 28% от суммы, превышающей указанную
Состоит в браке, совместное ведение хозяйства	15% с первых $29 750 плюс 28% от суммы, превышающей указанную
Состоит в браке, раздельное ведение хозяйства	15% с первых $14 875 плюс 28% от суммы, превышающей указанную

Например, одинокий работник, получающий облагаемый налогом доход в $20 000, платит налоги в сумме 0.15 × $17 850 + 0.28 × ($20 000 - $17 850). Напишите программу, которая позволяет пользователю указать категорию и облагаемый налогом доход, после чего вычисляет сумму налога. Используйте цикл, чтобы пользователь мог вводить разные варианты налогообложения.

11. Компания ABC Mail Order Grocery продает артишоки по цене $2.05 за фунт, свеклу по $1.15 за фунт и морковь по $1.09 за фунт. До добавления затрат на доставку компания предоставляет скидку 5% на заказы на сумму $100 и выше. Затраты составляют $6.50 за доставку и обработку заказа весом в 5 фунтов или менее, $14.00 за обработку и доставку заказа весом от 5 до 20 фунтов и $14.00 плюс $0.50 за каждый фунт для доставки заказа с весом, превышающим 20 фунтов. Напишите программу, которая использует оператор `switch` в цикле так, что в ответ на ввод `a` пользователь получает возможность указать желаемый вес артишоков в фунтах; в ответ на ввод `b` — вес свеклы в фунтах; в ответ на ввод `c` — вес моркови в фунтах; а в ответ на ввод `q` — завершить процесс заказа. Программа должна вести учет сумм нарастающим итогом. То есть если пользователь вводит 4 фунта свеклы и позже вводит еще 5 фунтов свеклы, программа должна сообщать о заказе 9 фунтов свеклы. Затем программа должна вычислить общие затраты, скидку, если есть, расходы на доставку и полную сумму заказа. Далее программа должна отобразить всю информацию о покупке: стоимость фунта товара, количество заказанных фунтов, стоимость каждого заказанного вида овощей, общую стоимость заказа, скидку (если есть), затраты на доставку и итоговую сумму заказа с учетом всех затрат.

8

Символьный ввод-вывод и проверка достоверности ввода

В ЭТОЙ ГЛАВЕ...

- Дополнительные сведения о вводе, выводе и различия между буферизированным и небуферизированным вводом
- Моделирование условия конца файла с клавиатуры
- Использование перенаправления для подключения программы к файлам
- Создание более дружественных пользовательских интерфейсов

В мире вычислений мы используем слова *ввод* и *вывод* многими путями. Мы обсуждаем устройства ввода и вывода, такие как клавиатуры, устройства USB, сканеры и лазерные принтеры. Мы говорим о данных, применяемых для ввода и вывода. Мы упоминаем функции, которые выполняют ввод и вывод. В этой главе основное внимание уделяется функциям ввода-вывода.

Функции ввода-вывода перемещают информацию в программу и из нее; примерами могут быть `printf()`, `scanf()`, `getchar()` и `putchar()`. Вы уже сталкивались с этими функциями в предшествующих главах, а теперь вы ознакомитесь с концепциями, лежащими в их основе. Наряду с этим вы увидите, как улучшить интерфейс между пользователем и программой.

Первоначально функции ввода-вывода не были частью определения языка С. Их разработка была оставлена за реализациями. На практике моделью для этих функций служила реализация С для операционной системы Unix. Учитывая весь прежний опыт, библиотека ANSI С содержит большое количество таких функций ввода-вывода, ориентированных на Unix, включая те, что мы использовали ранее. Поскольку эти стандартные функции должны работать с широким разнообразием компьютерных сред, они редко извлекают преимущества из возможностей, присущих конкретной системе. Поэтому многие поставщики реализаций языка С предлагают дополнительные функции ввода-вывода, которые задействуют специальные средства оборудования. Другие функции или семейства функций включаются в отдельные операционные системы, которые поддерживают, например, специальные графические интерфейсы вроде предоставляемых в Windows и Macintosh. Эти специализированные нестандартные функции позволяют писать программы, которые эксплуатируют конкретный компьютер более эффективно. К сожалению, часто они не могут применяться в других компьютерных системах.

Таким образом, мы сосредоточимся на стандартных функциях ввода-вывода, доступных для всех систем, т.к. они позволяют разрабатывать переносимые программы, которые можно легко перемещать из одной системы в другую. Они также стимулируют использование в программах файлов для ввода и вывода.

Многие программы сталкиваются с одной важной задачей — проверкой допустимости входных данных, т.е. с выяснением, ввел ли пользователь данные, которые ожидаются программой. В этой главе рассматриваются некоторые проблемы и решения, связанные с проверкой допустимости вводимых данных.

Односимвольный ввод-вывод: getchar() и putchar()

Как было показано в главе 7, функции `getchar()` и `putchar()` выполняют ввод и вывод по одному символу за раз. Такой подход может показаться нерациональным. В конце концов, можно легко читать группы, состоящие из нескольких символов, но этот метод вполне вписывается в возможности компьютера. Более того, такой подход лежит в основе большинства программ, имеющих дело с текстом — т.е. с обычными словами. Чтобы напомнить, как работают эти функции, в листинге 8.1 приведен очень простой пример. Здесь всего лишь принимаются символы из клавиатурного ввода и затем отображаются на экране. Такой процесс называется *эхо-выводом ввода*. В коде применяется цикл `while`, который завершается при обнаружении символа `#`.

Листинг 8.1. Программа echo.c

```
/* echo.c -- повторяет ввод */
#include <stdio.h>
int main(void)
{
   char ch;
   while ((ch = getchar()) != '#')
      putchar(ch);
      return 0;
}
```

Со времен появления стандарта ANSI C в языке с использованием функций getchar() и putchar() ассоциирован заголовочный файл stdio.h, потому он и был включен в программе. (Обычно getchar() и putchar() не являются истинными функциями, а определены с применением макросов препроцессора, как будет раскрыто в главе 16.) Выполнение программы приводит к обмену следующего вида:

```
Здравствуйте. Я хотел бы[enter]
Здравствуйте. Я хотел бы
приобрести #3 мешка картофеля.[enter]
приобрести
```

После наблюдения за работой этой программы может возникнуть вопрос, почему нужно набирать строку полностью, прежде чем введенные символы будут повторены на экране. Вас также может интересовать, есть ли лучший способ завершения ввода. Использование для завершения ввода специального символа, такого как #, предотвращает его употребление в тексте. Чтобы ответить на эти вопросы, давайте посмотрим, каким образом программы на языке C обрабатывают клавиатурный ввод. В частности, мы выясним, что собой представляет буферизация, и ознакомимся с понятием стандартного входного файла.

Буферы

Если вы запустите предыдущую программу на некоторых более старых системах, то вводимый текст может отображаться на экране немедленно. То есть выполнение этой программы могло бы дать примерно такой результат:

```
ЗЗддрраавввссттввууййттее..  ЯЯ  ххооттеелл  ббыы[enter]
ппррииообббрреессттии  #
```

Такое поведение является исключением. В большинстве систем ничего не произойдет до тех пор, пока не будет нажата клавиша <Enter>, как в первом примере запуска. Немедленный эхо-вывод вводимых символов на экране представляет собой пример *небуферизированного* (или *прямого*) ввода, при котором набираемые символы немедленно становятся доступным для ожидающей их программы. С другой стороны, задержанный эхо-вывод иллюстрирует *буферизированный* ввод, когда введенные символы накапливаются и хранятся во временной области, называемой *буфером*. Нажатие клавиши <Enter> делает блок набранных символов доступным для программы. Эти две разновидности ввода сравниваются на рис. 8.1.

Зачем иметь буферы? Во-первых, передача нескольких символов в виде блока является менее затратной по времени, чем отправка символов по одному. Во-вторых, в случае опечатки можно воспользоваться средствами корректировки, поддерживаемыми клавиатурой, и исправить опечатку. Затем после финального нажатия <Enter> программе будет передана исправленная версия.

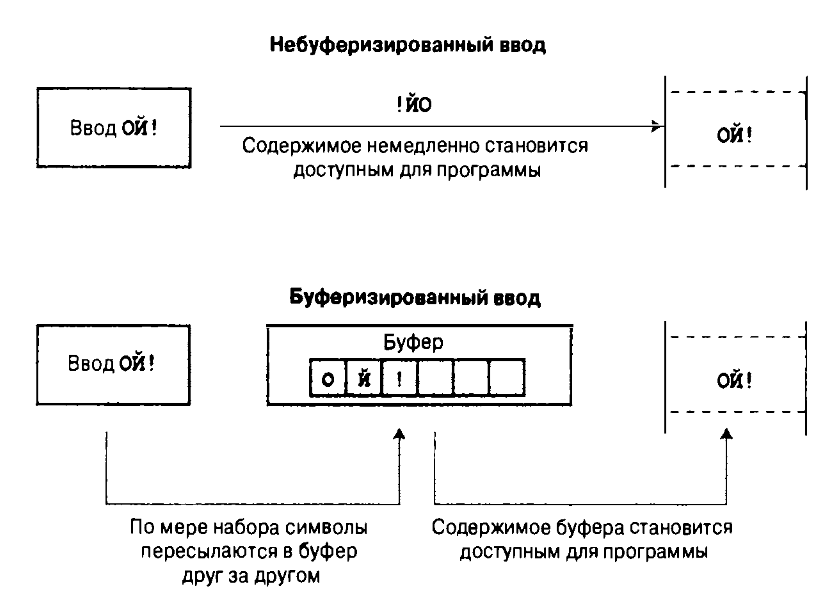


Рис. 8.1. Буферизированный и небуферизированный ввод

С другой стороны, небуферизированный ввод желателен для некоторых интерактивных программ. Например, в играх требуется, чтобы каждая команда выполнялась сразу же после нажатия клавиши. Таким образом, и буферизированный, и небуферизированный ввод имеют свои применения.

Существуют два вида буферизации — *полностью буферизированный* ввод-вывод и *построчно буферизированный* ввод-вывод. При полностью буферизированном вводе буфер сбрасывается (его содержимое отправляется в место назначения), когда он полон. Буферизация такого вида обычно происходит при файловом вводе. Размер буфера зависит от системы, но наиболее распространены значения 512 и 4096 байтов. В случае построчно буферизированного ввода-вывода буфер сбрасывается всякий раз, когда появляется символ новой строки. Клавиатурный ввод обычно является построчно буферизированным, так что нажатие <Enter> вызывает сброс буфера.

Каким типом ввода вы располагаете — буферизированным или небуферизированным? В ANSI C и последующих стандартах С указано, что ввод должен быть буферизированным, но в K&R C выбор изначально возлагался на разработчика компилятора. Тип ввода, используемый в системе, можно определить, запустив на выполнение программу `echo.c` и проанализировав ее поведение.

Причина того, что в ANSI C было принято решение считать стандартом буферизированный ввод, связана с тем, что некоторые компьютерные системы не разрешают небуферизированный ввод. Если ваш компьютер допускает небуферизированный ввод, то весьма вероятно, что применяемый вами компилятор С предлагает небуферизированный ввод в качестве опции. Например, многие компиляторы для компьютеров, совместимых с IBM PC, предоставляют специальное семейство функций, которые поддерживаются заголовочным файлом `conio.h` и предназначены для небуферизированного ввода. К их числу относятся функция `getche()` для небуферизированного ввода с эхо-выводом и функция `getch()` для небуферизированного ввода без эхо-вывода. (*Ввод с эхо-выводом* означает, что вводимый символ отображается на экране, а *ввод без эхо-вывода* — что нажатия клавиш не приводят отображению символов на экране.) В системах Unix используется другой подход, при котором буферизацией управляет сама система

Unix. В Unix вы применяете функцию ioctl() (которая входит в состав библиотеки Unix, но не является частью стандарта C) для указания желаемого типа ввода, после чего функция getchar() ведет себя должным образом. В ANSI C функции setbuf() и setvbuf() (глава 13) предоставляют определенный контроль над буферизацией, но присущие ряду систем ограничения снижают их эффективность. Выражаясь кратко, не существует способа, соответствующего стандарту ANSI, для обеспечения небуферизированного ввода; такие средства зависят от компьютерной системы. В этой книге мы предполагаем, что вы используете буферизированный ввод.

Завершение клавиатурного ввода

Программа echo.c останавливается, когда введен символ #, что удобно до тех пор, пока этот символ исключен из обычных входных данных. Однако, как уже было показано, символ # может встречаться и в обычном вводе. В идеальном случае хотелось бы иметь символ завершения ввода, который в обычном тексте не встречается. Такой символ не может неожиданно появиться в середине входного текста, останавливая программу раньше, чем планировалось. В C имеется ответ на эту потребность, но чтобы понять его, необходимо знать, как в C работать с файлами.

Файлы, потоки и ввод данных с клавиатуры

Файл — это область памяти, в которой хранится информация. Обычно файл размещается в постоянной памяти определенного вида, такого как жесткий диск, флэш-накопитель USB или оптический диск вроде DVD. Важность файлов для компьютерных систем не вызывает сомнений. Например, ваши программы на C хранятся в файлах, то же самое можно сказать о программах, применяемых для компиляции этих программ. Последний пример указывает на то, что некоторым программам требуется возможность доступа к отдельным файлам. При компиляции программы, хранящейся в файле echo.c, компилятор открывает этот файл и читает его содержимое. После завершения компиляции файл закрывается. Другие программы, такие как текстовые процессоры, не только открывают, читают и закрывают файлы, но также и записывают в них.

В C, как у мощного, гибкого и т.д. языка, имеется много библиотечных функций, предназначенных для открытия, чтения, записи и закрытия файлов. На одном уровне он может иметь дело с файлами, используя для этого базовые инструменты для работы с файлами из операционной системы. Это называется *низкоуровневым вводом-выводом*. Из-за многочисленных отличий между компьютерными системами создать стандартную библиотеку универсальных функций для низкоуровневого ввода-вывода невозможно, и в стандарте ANSI C такая попытка даже не предпринимается. Тем не менее, язык C также умеет работать с файлами на другом уровне, который имеет название *стандартный пакет ввода-вывода*. При этом предполагается создание стандартной модели и стандартного набора функций ввода-вывода, предназначенных для работы с файлами. На таком более высоком уровне различия между системами поддерживаются специфическими реализациями C, так что вы имеете дело с унифицированным интерфейсом.

А о каких отличиях между компьютерными системами идет речь? Например, разные системы сохраняют файлы по-разному. Некоторые хранят содержимое файла в одном месте, а информацию о нем — в другом. Одни системы встраивают описание файла в сам файл. При работе с текстами многие системы для обозначения конца строки применяют одиночный символ новой строки. Другие могут использовать для этого комбинацию символов возврата каретки и перевода строки. Некоторые системы измеряют размер файлов до ближайшего байта, а другие — в блоках байтов.

Когда вы применяете стандартный пакет ввода-вывода, вы защищены от воздействия таких отличий. Следовательно, для проверки на предмет символа новой строки можно использовать конструкцию `if (ch == '\n')`. Если система применяется комбинация символов возврата каретки и перевода строки, то функции ввода-вывода выполняют автоматическую трансляцию между двумя этими представлениями в обоих направлениях.

Концептуально программа на С имеет дело с потоком, а не напрямую с файлом. *Поток* – это идеализированное течение данных, на которое отображается действительный ввод или вывод. Это означает, что разнообразные виды ввода с отличающимися свойствами представлены с помощью потоков, имеющих более унифицированные свойства. Тогда процесс открытия файла становится процессом ассоциирования потока с файлом, а чтение и запись осуществляются через поток.

Файлы подробно обсуждаются в главе 13. Для целей настоящей главы просто запомните, что в языке С устройства ввода и вывода трактуются таким же образом, как обычные файлы на устройствах хранения. В частности, клавиатура и устройство отображения считаются файлами, автоматически открываемыми каждой программой на С. Клавиатурный ввод представлен потоком по имени `stdin`, а вывод на экран (или на телетайп либо другое устройство вывода) представлен потоком по имени `stdout`. Функции `getchar()`, `putchar()`, `printf()` и `scanf()` являются членами стандартного пакета ввода-вывода, и все они имеют дело с двумя упомянутыми потоками.

Одно из следствий всего этого заключается в том, что при работе с клавиатурным вводом можно использовать те же самые приемы, как и при работе с файлами. Например, программе, читающей файл, необходим способ обнаружения конца файла, чтобы знать, где останавливать чтение. Поэтому функции для ввода в С оснащены встроенным средством обнаружения конца файла. Поскольку клавиатурный ввод трактуется подобно файлу, вы должны иметь возможность применять это средство обнаружения конца файла также и в данном случае. Давайте посмотрим, как это делается, начав с файлов.

Конец файла

Операционная система нуждается в каком-то способе для выяснения, где начинается и где заканчивается каждый файл. Один из методов обнаружения конца файла предусматривает помещение в файл специального символа, помечающего его конец. В свое время такой метод использовался, к примеру, в текстовых файлах в средах операционных систем CP/M, IBM-DOS и MS-DOS. Теперь эти операционные системы для пометки конца файла могли бы применять встроенный символ <Ctrl+Z>. Когда-то это было единственным средством, которое использовали операционные системы, но сейчас доступны другие варианты наподобие отслеживания размера файла. Таким образом, современный текстовый файл может содержать, а может и не содержать встроенный символ <Ctrl+Z>, однако если он присутствует, операционная система будет трактовать его как маркер конца файла. Этот подход иллюстрируется на рис. 8.2.

Фраза:

Робот Бишоп
плавно открыл люк
и ответил на свой вызов.

Фраза в файле:

```
Робот Бишоп\nплавно открыл люк\nи ответил на свой вызов.\n^Z
```

Рис. 8.2. Файл с маркером конца файла

Второй подход заключается в том, что операционная система хранит информацию о размере файла. Если файл содержит 3000 байтов, а программа прочитала 3000 байтов, значит, она достигла конца файла. Операционная система MS-DOS и ей подобные применяют этот подход для двоичных файлов, т.к. данный метод позволяет хранить в файлах все символы, в том числе <Ctrl+Z>. Более новые версии DOS также используют этот подход для текстовых файлов. В Unix он применяется ко всем файлам.

Такое многообразие методов поддерживается в С за счет того, что функция `getchar()` возвращает специальное значение при достижении конца файла независимо от того, как в действительности конец файла обнаруживается операционной системой. Этому специальному значению было назначено имя `EOF` ("end of file" – "конец файла"). Следовательно, возвращаемым значением функции `getchar()` в случае обнаружения конца файла является `EOF`. Функция `scanf()` при обнаружении конца файла также возвращает `EOF`. Обычно `EOF` определяется в файле `stdio.h` следующим образом:

```
#define EOF (-1)
```

Почему было выбрано значение `-1`? Обычно функция `getchar()` возвращает значение в диапазоне от `0` до `127`, поскольку они соответствуют стандартному набору символов, но она может возвращать значения от `0` до `255`, если система распознает расширенный набор символов. В любом случае значение `-1` не относится ни к одному из символов, так что оно может использоваться для сообщения о конце файла.

В некоторых системах `EOF` может быть определено как значение, не равное `-1`, но это определение всегда отличается от возвращаемого значения, генерируемого допустимым входным символом. Если вы включили файл `stdio.h` и применяете символическую константу `EOF`, то не обязаны беспокоиться о ее числовом определении. Важно понимать, что `EOF` представляет значение, сигнализирующее об обнаружении конца файла, а не значение, действительно находящееся файле.

А как использовать `EOF` в программе? Нужно сравнить возвращаемое значение `getchar()` с `EOF`. Если они отличаются, конец файла пока еще не достигнут. Другими словами, можно указывать выражение, подобное следующему:

```
while ((ch = getchar()) != EOF)!
```

Что, если производится чтение клавиатурного ввода, а не файла? Большинство систем (но не все) поддерживают какой-то способ эмулировать условие конца файла с помощью клавиатуры. С учетом этого базовую программу чтения и эхо-вывода можно переписать, как показано в листинге 8.2.

Листинг 8.2. Программа `echo_eof.c`

```
/* echo_eof.c — повторяет на экране ввод до достижения конца файла */
#include <stdio.h>
int main(void)
{
   int ch;

   while ((ch = getchar()) != EOF)
      putchar(ch);

   return 0;
}
```

Обратите внимание на перечисленные ниже аспекты.

- Вам не придется определять EOF, т.к. об этом уже позаботился заголовочный файл stdio.h.
- Вам не нужно беспокоиться о действительном значении EOF, поскольку оператор #define в файле stdio.h позволяет иметь дело с символическим представлением EOF. Вы не должны писать код, в котором предполагается, что EOF имеет какое-то конкретное значение.
- Тип переменной ch изменен с char на int, потому что переменные типа char могут быть представлены целочисленными значениями без знака в диапазоне от 0 до 255, но EOF может иметь числовое значение -1. Такое значение недопустимо для типа char без знака, но допустимо для типа int. К счастью, функция getchar() сама имеет тип int, поэтому она может читать символ EOF. В реализациях, поддерживающих тип char со знаком, можно обойтись объявлением переменной ch как имеющей тип char, но лучше применить более общую форму.
- Именно из-за того, что функция getchar() имеет тип int, некоторые компиляторы предупреждают о возможной потере данных при присваивании возвращаемого значения этой функции переменной типа char.
- Тот факт, что переменная ch является целочисленной, никак не беспокоит функцию putchar(). Она по-прежнему выводит символьный эквивалент.
- Чтобы использовать эту программу с клавиатурным вводом, необходим какой-то способ набора символа EOF. Понятно, что нельзя просто набрать буквы *E O F* или ввести -1. (Ввод с клавиатуры -1 приводит к передаче в программу двух символов: дефиса и цифры 1.) Взамен понадобится выяснить, какой символ требует система. Например, в большинстве систем Unix и Linux нажатие комбинации клавиш <Ctrl+D> в *начале* строки вызывает передачу сигнала конца файла. Многие системы в качестве сигнала конца файла распознают комбинацию <Ctrl+Z> в начале строки, а некоторые интерпретируют ее как таковую в любом месте строки.

Ниже показан пример применения буферизированного ввода в программе echo_eof.c под управлением Unix:

```
У него не было даже пальто.
У него не было даже пальто.
  В город молодой человек вошел в зеленом в талию костюме.
  В город молодой человек вошел в зеленом в талию костюме.
                        И. Ильф, Е. Петров
                        И. Ильф, Е. Петров
[Ctrl+D]
```

Каждый раз, когда вы нажимаете <Enter>, хранящиеся в буфере символы обрабатываются, и копия строки выводится. Это продолжается вплоть до эмуляции конца файла в стиле Unix. В другой системе пришлось бы нажать комбинацию <Ctrl+Z>.

Давайте подумаем о возможностях, доступных для программы echo_eof.c. Она копирует на экран любые переданный ей ввод. Предположим, что вы каким-то образом предоставили ей файл. Тогда программа выведет на экран содержимое этого файла, остановившись по достижении конца файла после обнаружения сигнала EOF. А еще представим, что вместо этого вы нашли способ направить вывод программы в файл. Тогда можно было ввести данные с клавиатуры и использовать echo_eof.c для их сохранения в файле. Далее предположим, что вам удалось сделать то и другое одновременно: направить ввод из одного файла в echo_eof.c и переслать вывод в дру-

гой файл. Тогда программу echo_eof.c можно было бы применять для копирования файлов. Эта небольшая программа обладает потенциалом для просмотра содержимого файлов, создания новых файлов и копирования существующих файлов – весьма неплохо для такой короткой программы! Ключом является управление потоком ввода и вывода, что и будет следующей рассматриваемой темой.

НА ЗАМЕТКУ! Эмулированный символ EOF и графические интерфейсы

Концепция эмуляции символа EOF возникла в среде командной строки, использующей текстовый интерфейс. В такой среде пользователь взаимодействует с программой через нажатия клавиш, и сигнал EOF генерирует операционная система. Некоторые действия не очень хорошо транслируются в графические среды, такие как Windows и Macintosh, с более сложными пользовательскими интерфейсами, которые включают перемещение курсора мыши и щелчки на кнопках. Поведение программы, сталкивающейся с эмулированным символом EOF, зависит от компилятора и типа проекта. Например, в зависимости от настроек нажатие <Ctrl+Z> может завершить ввод данных или же завершить выполнение самой программы.

Перенаправление и файлы

С вводом и выводом связаны функции, данные и устройства. Рассмотрим для примера программу echo_eof.c. В ней используется функция getchar(). Входным устройством (согласно нашему предположению) является клавиатура, а поток входных данных состоит из отдельных символов. Представим, что вы хотите сохранить ту же самую функцию ввода и ту же разновидность данных, но изменить место, где программа ожидает найти данные. При этом возникает вопрос: а как программа узнает, откуда получать ввод?

По умолчанию программа на С, в которой применяется стандартный пакет ввода-вывода, считает источником входных данных стандартный ввод. Это поток, ранее идентифицированный как stdin. Он представляет собой все, что было настроено в качестве обычного метода чтения данных в компьютер. Им могут быть такие устаревшие устройства, как магнитная лента, перфокарты или телетайп, либо (что мы будем подразумевать в дальнейшем) клавиатура или какая-то передовая технология наподобие голосового ввода. Однако в современных компьютерах этот поток является настраиваемым инструментом, который можно ориентировать на какое-то другое место. В частности, программе можно указать на необходимость получения ввода из файла, а не с клавиатуры.

Существуют два способа заставить программу работать с файлами. Один из них предполагает явное использование специальных функций, которые открывают, закрывают, читают, записывают в файлы и т.д. Исследование этого метода мы отложим до главы 13. Второй способ заключается в применении программы, спроектированной для работы с клавиатурой и экраном, но перенаправлении ввода и вывода в разные каналы – например, в файл и из файла. Другими словами поток stdin переназначается в файл. Функция getchar() продолжает получать данные из потока, в действительности не интересуясь, откуда поток получает свои данные. Такой подход (перенаправление) в некоторых аспектах является более ограниченным, чем первый подход, но он намного проще в использовании и позволяет ознакомиться с распространенными приемами обработки файлов.

Одна из главных проблем перенаправления состоит в том, что оно связано с операционной системой, а не с языком С. Однако многие среды с языком С, включая Unix, Linux и режим командной строки Windows, поддерживают перенаправление, а некоторые реализации С эмулируют его в системах, где перенаправление отсутствует. Операционная система Apple OS X действует поверх Unix, и режим командной строки

Unix можно инициировать, запустив приложение Terminal. Мы взглянем на версии перенаправления в Unix, Linux и Windows.

Перенаправление в Unix, Linux и командной строке Windows

Операционные системы Unix (в режиме командной строки), Linux (аналогично) и режим командной строки Windows (который имитирует старую среду командной строки DOS) позволяют перенаправлять как ввод, так и вывод. Перенаправление ввода предоставляет программе возможность применять для ввода файл вместо клавиатуры, а перенаправление вывода – использовать для вывода файл вместо экрана.

Перенаправление ввода

Предположим, что вы скомпилировали программу echo_eof.c и поместили ее исполняемую версию в файл по имени echo_eof (или echo_eof.exe в системе Windows). Чтобы запустить программу, введите имя файла:

```
echo_eof
```

Программа выполняется так, как было описано ранее, получая ввод с клавиатуры. Теперь предположим, что вы хотите применить эту программу к текстовому файлу с именем words. *Текстовый файл* содержит текст, т.е. данные хранятся в виде символов, воспринимаемых человеком. Например, это может быть очерк или программа на языке С. Файл, содержащий инструкции машинного языка, такой как файл с исполняемой версией программы, не является текстовым. Поскольку программа работает с символами, она должна использоваться с текстовыми файлами. Все, что понадобится – ввести следующую команду:

```
echo_eof < words
```

Символ < представляет собой операцию перенаправления в Unix, Linux и DOS/Windows. Она приводит к тому, что файл words ассоциируется с потоком stdin с передачей по каналу содержимого файла в программу echo_eof. Сама программа echo_eof даже не знает (и не беспокоится об этом), что ввод поступает из файла, а не с клавиатуры. Ей известен только тот факт, что ей поставляется поток символов, поэтому программа читает и выводит по одному символу за раз, пока не будет достигнут конец файла. Поскольку в С файлы и устройства ввода-вывода приравнены друг к другу, файл теперь является *устройством* ввода-вывода. Опробуйте это!

НА ЗАМЕТКУ! Дополнительные пояснения по перенаправлению

В Unix, Linux и командной строке Windows пробелы с обеих сторон знака < не обязательны. Некоторые системы вроде AmigaDOS (упоминается специально для тех, кто ностальгирует за старыми добрыми временами) поддерживают перенаправление, но не разрешают указывать пробел между символом перенаправления и именем файла.

Ниже приводится пример запуска программы echo_eof с конкретным текстовым файлом words; знак $ – одно из стандартных приглашений на ввод в Unix и Linux. В Windows/DOS приглашение на ввод может выглядеть как A> или C>.

```
$ echo_eof < words
Пешеходов надо любить.
Пешеходы составляют большую часть человечества.
Мало того – лучшую его часть.
Пешеходы создали мир.
$
```

Итак, перейдем к сути дела.

Перенаправление вывода

Теперь предположим, что вы хотите, чтобы программа `echo_eof` пересылала клавиатурный ввод в файл по имени `mywords`. В этом случае потребуется ввести следующую команду и начать набор:

```
echo_eof > mywords
```

Знак > представляет еще одну операцию перенаправления. Он приводит к созданию нового файла с именем `mywords` и переадресует в него вывод `echo_eof` (т.е. копию символов, набираемых на клавиатуре). Перенаправление переназначает `stdout` с устройства отображения (экрана) на файл `mywords`. Если файл `mywords` уже существует, обычно он очищается и заменяется новым содержимым. (Однако многие операционные системы предоставляют возможность защиты существующих файлов, делая их файлами только для чтения.) Все, что вы видите на экране — это символы, набираемые на клавиатуре, а их копии поступают в файл. Чтобы завершить программу, нажмите комбинацию клавиш <Ctrl+D> (Unix) или <Ctrl+Z> (DOS) в начале строки. Попробуйте это сами. Если не можете придумать, что вводить с клавиатуры, просто повторите приведенный ниже пример. В нем присутствует приглашение `$` системы Unix. Не забывайте завершать каждую строку нажатием <Enter>, чтобы содержимое буфера отправлялось в программу.

```
$ echo_eof > mywords
У вас не должно возникать никаких проблем с запоминанием того, что делает
тот или иной оператор перенаправления. Запомните только, что оператор
указывает направление потока информации. Представьте себе, что
это воронка.
[Ctrl+D]
$
```

После того, как комбинация <Ctrl+D> или <Ctrl+Z> будет обработана, программа завершится и на экране снова отобразится приглашение на ввод. Выполнила ли программа свою работу? Команда `ls` системы Unix или команда `dir` командной строки Windows, которые выводят на экран список имен файлов, должны подтвердить существование файла `mywords`. Вы можете воспользоваться командой `cat` в Unix и Linux или `type` в DOS для проверки его содержимого либо запустить программу `echo_eof` снова, на этот раз перенаправив файл `mywords` в программу:

```
$ echo_eof < mywords
У вас не должно возникать никаких проблем с запоминанием того, что делает
тот или иной оператор перенаправления. Запомните только, что оператор
указывает направление потока информации. Представьте себе, что
это воронка.
$
```

Комбинированное перенаправление

Теперь предположим, что вы хотите создать копию файла `mywords` и назначить ей имя `savewords`. Достаточно ввести следующую команду:

```
echo_eof < mywords > savewords
```

и дело сделано. Показанная далее команда также сработает, потому что порядок указания операций перенаправления не играет роли:

```
echo_eof > savewords < mywords
```

Однако будьте внимательны: не применяйте один и тот же файл для ввода и вывода внутри одной команды:

```
echo_eof < mywords > mywords   ← НЕПРАВИЛЬНО!
```

Причина в том, что конструкция > mywords приводит к усечению исходного файла mywords до нулевой длины до того, как он будет использован в качестве ввода.

В двух словах, существуют правила, регламентирующие применение операций перенаправления (< и >) в средах Unix, Linux и Windows/DOS.

- Операция перенаправления соединяет *исполняемую* программу (равно как и стандартные команды операционной системы) с файлом данных. Она не может использоваться для соединения одного файла данных с другим, а также для соединения одной программы с другой.
- С помощью этих операций ввод нельзя получать из более чем одного файла, а вывод направлять в более чем один файл.
- Обычно пробелы между именами и операциями являются необязательными за редким исключением, когда применяются специальные символы, имеющие особый смысл в командной оболочке Unix или Linux либо в режиме командной строки Windows. Например, можно было бы иметь команду echo_eof < words.

Вы уже видели несколько правильных примеров. Ниже перечислен ряд некорректных примеров, в которых addup и count выступают как исполняемые программы, а fish и beets — как текстовые файлы:

```
fish > beets           ← Нарушает первое правило
addup < count          ← Нарушает первое правило
addup < fish < beets   ← Нарушает второе правило
count > beets fish     ← Нарушает второе правило
```

В средах Unix, Linux и Windows/DOS также доступна операция >>, которая позволяет добавлять данные в конец существующего файла, и операция конвейера (|), делающая возможным соединение вывода одной программы с вводом другой программы. За дополнительными сведениями по всем этим операциям обращайтесь к соответствующим книгам.

Комментарии

Перенаправление позволяет использовать программы, предназначенные для обработки ввода с клавиатуры, с файлами. Для этого в программе должна предприниматься проверка на предмет достижения конца файла. Например, в главе 7 была представлена программа, которая подсчитывала слова до появления первого символа |. Измените тип ch с char на int и замените '|' на EOF в выражении проверки цикла, после чего эту программу можно будет применять для подсчета слов в текстовых файлах.

Перенаправление — это концепция командной строки, т.к. оно задается путем ввода с клавиатуры специальных символов в командной строке. Если вы не используете среду командной строки, возможность применения этого приема по-прежнему доступна. Во-первых, в некоторых интегрированных средах имеются пункты меню, позволяющие указывать перенаправление. Во-вторых, в системах Windows можно открыть окно командной строки и запустить исполняемый файл в командной строке. По умолчанию среда Microsoft Visual Studio помещает исполняемый файл в подпапку Debug внутри папки проекта. Имя файла будет иметь то же базовое имя, что у проекта, и расширение .exe. По умолчанию система XCode также называет исполняемый файл по имени проекта и помещает его в папку Debug. Исполняемый файл можно запустить из

утилиты Terminal, которая запускает соответствующую версию Unix. Однако в случае использования этой утилиты вероятно проще применять один из компиляторов командной строки (GCC или Clang), который можно загрузить из веб-сайта Apple.

Если перенаправление не работает, можете попытаться заставить программу открыть файл напрямую. В листинге 8.3 показан пример с минимальными пояснениями. Более подробную информацию можно найти в главе 13. Предназначенный для чтения файл должен находиться в том же каталоге, что и исполняемый файл.

Листинг 8.3. Программа `file_eof.c`

```
// file_eof.c -- открывает файл и отображает его содержимое
#include <stdio.h>
#include <stdlib.h>         // для функции exit()
int main()
{
   int ch;
   FILE * fp;
   char fname[50];          // для хранения имени файла

   printf("Введите имя файла: ");
   scanf("%s", fname);
   fp = fopen(fname, "r");  // открыть файл для чтения
   if (fp == NULL)          // попытка завершилась неудачей
   {
      printf("Не удается открыть файл. Программа завершена.\n");
      exit(1);              // выйти из программы
   }
   // функция getc(fp) получает символ из открытого файла
   while ((ch = getc(fp)) != EOF)
      putchar(ch);
   fclose(fp);              // закрыть файл

   return 0;
}
```

Сводка: перенаправление ввода и вывода

В большинстве систем с языком С перенаправление можно использовать либо для всех программ через операционную систему, либо только для программ на С посредством возможностей, предоставляемых компилятором С. В следующих примерах `prog` — это имя исполняемой программы, а `file1` и `file2` — имена файлов.

Перенаправление вывода в файл (>)

```
prog >file1
```

Перенаправление ввода из файла (<)

```
prog <file2
```

Комбинированное перенаправление

```
prog <file2 >file1
prog >file1 <file2
```

В обеих формах `file2` применяется для ввода и `file1` для вывода.

Пробелы

Некоторые системы требуют наличие пробела слева от знака операции перенаправления и отсутствие пробела справа от этого знака. Другие системы (например, Unix) допускают наличие пробелов с обеих сторон либо или их отсутствие.

Создание дружественного пользовательского интерфейса

Большинству из нас приходилось писать программы, пользоваться которыми было не особенно удобно. К счастью, в C имеются инструменты для превращения ввода в более гладкий и приятный процесс. К сожалению, изучение этих инструментов поначалу порождает новые проблемы. Цель настоящего раздела в том, чтобы помочь решить часть проблем, препятствующих созданию более дружественного пользовательского интерфейса, который облегчает ввод интерактивных данных и минимизирует эффект от ошибочного ввода данных.

Работа с буферизированным вводом

Буферизированный ввод часто удобен для пользователя, т.к. он предоставляет возможность редактирования входных данных до отправки их в программу, но для программиста он может стать источником дополнительных забот, когда задействован символьный ввод. Как можно было заметить в ряде приводимых ранее примеров, проблема заключается в том, что буферизированный ввод требует нажатия клавиши <Enter> для передачи введенных данных. Это действие пересылает также символ новой строки, который программа должна обработать. Давайте исследуем эту и другие проблемы на примере программы угадывания чисел. Вы выбираете число, а компьютер пытается его угадать. В программе применяется довольно скучный метод, но мы сосредоточимся на вводе-выводе, а не на алгоритме. В листинге 8.4 приведена начальная версия программы, которая требует дальнейшей доработки.

Листинг 8.4. Программа guess.c

```
/* guess.c -- неэффективное и чреватое ошибками угадывание числа */
#include <stdio.h>
int main(void)
{
   int guess = 1;
   printf("Выберите целое число в интервале от 1 до 100. Я попробую угадать ");
   printf("его.\nНажмите клавишу y, если моя догадка верна и ");
   printf("\nклавишу n в противном случае.\n");
   printf("Вашим числом является %d?\n", guess);
   while (getchar() != 'y')      /* получить ответ, сравнить с y */
      printf("Ладно, тогда это %d?\n", ++guess);
   printf("Я знал, что у меня получится!\n");
   return 0;
}
```

Вот пример выполнения программы:

```
Выберите целое число в интервале от 1 до 100. Я попробую угадать его.
Нажмите клавишу y, если моя догадка верна и
клавишу n в противном случае.
Вашим числом является 1?
n
Ладно, тогда это 2?
Ладно, тогда это 3?
n
Ладно, тогда это 4?
Ладно, тогда это 5?
y
Я знал, что у меня получится!
```

Вопреки ожиданиям алгоритма, реализованного в программе, мы выбрали небольшое число. Обратите внимание на то, что после ввода n программа делает два предположения. Дело в том, что программа читает ответ n как отрицание того, что было загадано число 1, и затем считывает символ новой строки как отрицание того факта, что было загадано число 2.

Одно из решений предусматривает использование цикла while для отбрасывания остатка введенной строки, включая символ новой строки. Дополнительное достоинство такого подхода состоит в том, что ответы вроде no или no way будут трактоваться просто как n. Версия в листинге 8.4 интерпретирует no как два ответа. Ниже показан пример цикла, в котором эта проблема устранена:

```
while (getchar() != 'y')  /* получить ответ, сравнить с y */
{
   printf("Ладно, тогда это %d?\n", ++guess);
   while (getchar() != '\n')
      continue;           /* пропустить оставшуюся часть входной строки*/
}
```

В случае применения этого цикла получается следующий вывод:

```
Выберите целое число в интервале от 1 до 100. Я попробую угадать его.
Нажмите клавишу y, если моя догадка верна и
клавишу n в противном случае.
Вашим числом является 1?
n
Ладно, тогда это 2?
no
Ладно, тогда это 3?
no sir
Ладно, тогда это 4?
forget it
Ладно, тогда это 5?
y
Я знал, что у меня получится!
```

Проблема с символом новой строки решена. Тем не менее, вряд ли можно посчитать нормальным тот факт, что f трактуется как n. Для устранения этого дефекта можно воспользоваться оператором if, чтобы отфильтровать другие ответы. Прежде всего, определите переменную типа char для хранения ответа:

```
char response;
```

Затем внесите изменения в цикл, чтобы он приобрел следующий вид:

```
while ((response = getchar()) != 'y')   /* получить ответ */
{
   if (response == 'n')
      printf("Ладно, тогда это %d?\n", ++guess);
   else
      printf("Принимаются только варианты y или n.\");
   while (getchar() != '\n')
      continue;              /* пропустить оставшуюся часть входной строки*/
}
```

Теперь вывод выглядит так:

```
Выберите целое число в интервале от 1 до 100. Я попробую угадать его.
Нажмите клавишу y, если моя догадка верна и
клавишу n в противном случае.
Вашим числом является 1?
n
Ладно, тогда это 2?
no
Ладно, тогда это 3?
no sir
Ладно, тогда это 4?
forget it
Принимаются только варианты y или n.
n
Ладно, тогда это 5?
y
Я знал, что у меня получится!
```

При написании интерактивных программ вы должны стараться предвосхищать возможности нарушения инструкций пользователями. Программу необходимо проектировать так, чтобы она элегантно обрабатывала ошибки пользователей. Пользователей следует уведомить о допущенной ошибке и дать дополнительный шанс.

Разумеется, вы должны предоставить пользователю четкие инструкции, однако независимо от того, насколько они ясны, всегда найдутся те, кто интерпретирует их неправильно, а затем обвинит вас в составлении непонятных инструкций.

Смешивание числового и символьного ввода

Предположим, что программа требует символьного ввода с помощью `getchar()` и числового ввода посредством `scanf()`. Каждая из этих функций хорошо делает свою работу, но смешивать их нелегко. Причина в том, что функция `getchar()` читает каждый символ, включая пробелы, символы табуляции и новой строки, в то время как `scanf()` при чтении чисел пропускает пробелы, символы табуляции и новой строки.

Чтобы продемонстрировать проблемы, которые при этом возникают, в листинге 8.5 представлена программа, которая в качестве ввода считывает символ и два числа. Затем она выводит таблицу с этим символом, имеющую столько строк и столбцов, сколько было указано во введенных числах.

Листинг 8.5. Программа showchar1.c

```c
/* showchar1.c -- программа с крупной проблемой ввода-вывода */
#include <stdio.h>
void display(char cr, int lines, int width);
int main(void)
{
    int ch;              /* выводимый символ          */
    int rows, cols;      /* количество строк и столбцов */
    printf("Введите символ и два целых числа:\n");
    while ((ch = getchar()) != '\n')
    {
        scanf("%d %d", &rows, &cols);
        display(ch, rows, cols);
        printf("Введите еще один символ и два целых числа;\n");
        printf("для завершения введите символ новой строки.\n");
    }
    printf("Программа завершена.\n");
    return 0;
}
```

```
void display(char cr, int lines, int width)
{
   int row, col;

   for (row = 1; row <= lines; row++)
   {
      for (col = 1; col <= width; col++)
         putchar(cr);
      putchar('\n');  /* закончить строку и начать новую */
   }
}
```

Обратите внимание на то, что программа читает символ как тип int, чтобы сделать возможной проверку на EOF. Однако она передает этот символ функции display() как тип char. Поскольку char меньше int, некоторые компиляторы предупредят о преобразовании. В данном случае предупреждение можно проигнорировать. Или же вывод предупреждения можно предотвратить, добавив приведение типа:

```
display(char(ch), rows, cols);
```

Программа устроена так, что функция main() получает данные, а функция display() производит вывод. Давайте взглянем на результаты выполнения программы, чтобы увидеть, в чем заключается проблема:

```
Введите символ и два целых числа:
c 2 3
ccc
ccc
Введите еще один символ и два целых числа;
для завершения введите символ новой строки.
Программа завершена.
```

Сначала программа работает хорошо. Вы вводите c 2 3, а программа выводит две строки по три символа c, как и ожидалось. Затем она предлагает ввести следующий набор данных и завершает работу, прежде чем вы сможете ответить. Что пошло не так? Проблема снова с символом новой строки, на этот раз с тем, который находится непосредственно после числа 3 в первой введенной строке. Функция scanf() оставляет его во входной очереди. В отличие от scanf(), функция getchar() не пропускает символов новой строки, так что этот символ читается getchar() на следующей итерации цикла, прежде чем вы получите возможность ввести что-либо еще. Затем он присваивается переменной ch, а равенство ch символу новой строки означает завершение цикла.

Чтобы устранить эту проблему, программа должна пропускать любые символы новой строки или пробелы между последним числом, набранным в одном цикле ввода, и первым символом, набираемым в следующей строке. Кроме того, было бы неплохо, если бы в дополнение к проверке getchar() программу можно было прекратить на стадии выполнения функции scanf(). Все это реализовано в следующей версии программы, показанной в листинге 8.6.

Листинг 8.6. Программа showchar2.c

```
/* showchar2.c -- выводит символы в строках и столбцах */
#include <stdio.h>
void display(char cr, int lines, int width);
int main(void)
{
```

```
    int ch;              /* выводимый символ          */
    int rows, cols;      /* количество строк и столбцов */

    printf("Введите символ и два целых числа:\n");
    while ((ch = getchar()) != '\n')
    {
        if (scanf("%d %d",&rows, &cols) != 2)
            break;
        display(ch, rows, cols);
        while (getchar() != '\n')
        continue;
        printf("Введите еще один символ и два целых числа;\n");
        printf("для завершения введите символ новой строки.\n");
    }
    printf("Программа завершена.\n");
    return 0;
}
void display(char cr, int lines, int width)
{
    int row, col;

    for (row = 1; row <= lines; row++)
    {
        for (col = 1; col <= width; col++)
            putchar(cr);
        putchar('\n'); /* закончить строку и начать новую */
    }
}
```

Оператор `while` заставляет программу пропускать все символы, следующие за вводом `scanf()`, включая символ новой строки. Это подготавливает цикл для чтения первого символа в начале следующей строки. Другими словами, данные можно вводить без ограничений:

```
Введите символ и два целых числа:
c 1 2
cc
Введите еще один символ и два целых числа;
для завершения введите символ новой строки.
! 3 6
!!!!!!
!!!!!!
!!!!!!
Введите еще один символ и два целых числа;
для завершения введите символ новой строки.

Программа завершена.
```

За счет использования оператора `if` вместе с `break` мы завершаем выполнение программы, если значение, возвращаемое функцией `scanf()`, не равно 2. Это происходит, когда одно или оба входных значения не являются целыми числами или встретился символ конца файла.

Проверка допустимости ввода

На практике пользователи программ не всегда следуют инструкциям, и вполне может возникнуть несоответствие между тем, что программа ожидает в качестве ввода, и тем, что в действительности она получает. Такие условия могут привести к аварий-

ному завершению программы. Тем не менее, вероятные ошибки часто можно предугадать и, приложив дополнительные усилия по программированию, заставить программу обнаруживать их и должным образом обрабатывать.

Предположим для примера, что имеется цикл, обрабатывающий неотрицательные числа. Один из видов ошибок, которые может совершить пользователь — ввод отрицательного числа. Для проверки такой ситуации можно предусмотреть выражение отношения:

```
long n;
scanf("%ld", &n);     // получить первое значение
while (n >= 0)        // обнаружить значение, выходящее за пределы диапазона
{
   // обработать n
   scanf("%ld", &n); // получить следующее значение
}
```

Еще одна потенциальная ловушка связана с тем, что пользователь может ввести значение неподходящего типа, такое как символ q. Один из способов обнаружения такого вида ошибок предполагает проверку возвращаемого значения функции scanf(). Как вы помните, она возвращает количество успешно прочитанных элементов; таким образом, выражение

```
scanf("%ld", &n) == 1
```

будет истинным, только если пользователь вводит целое число. Это требует внесения в код следующего изменения:

```
long n;
while (scanf("%ld", &n) == 1 && n >= 0)
{
   // обработать n
}
```

Условие цикла while звучит так: "пока ввод является целочисленным значением и это значение положительно".

В последнем примере цикл прекращается, когда пользователь вводит значение некорректного типа. Однако программу можно сделать более дружественной к пользователю и предоставить ему возможность ввести значение правильного типа. В этом случае понадобится отбросить ввод, который привел scanf() к ошибке при первом вызове, т.к. эта функция оставляет неподходящие данные во входной очереди. Здесь пригодится тот факт, что ввод является потоком символов, поскольку можно воспользоваться функцией getchar() для посимвольного чтения. Все эти идеи можно даже реализовать в виде функции, как показано ниже:

```
long get_long(void)
{
   long input;
   char ch;
   while (scanf("%ld", &input) != 1)
   {
   while ((ch = getchar()) != '\n')
         putchar(ch);  // отбросить неправильный ввод
      printf(" не является целым числом.\nВведите ");
      printf("целое число, такое как 25, -178 или 3: ");
   }
   return input;
}
```

Функция get_long() пытается прочитать значение типа int в переменную input. Если ей это не удастся, происходит вход в тело внешнего цикла while. Затем во внутреннем цикле while выполняется посимвольное чтение проблемного ввода. Обратите внимание, что функция выбран вариант с отбрасыванием всего, что осталось во входной строке. Другим возможным вариантом может быть отбрасывание следующего символа или слова. Далее функция предлагает пользователю повторить попытку ввода. Внешний цикл продолжает выполняться до тех пор, пока пользователь успешно не введет целое число, что приведет к возврату scanf() значения 1.

После того, как пользователь преодолеет все препятствия, не позволяющие ему вводить целые числа, программа может выяснить, допустимы ли введенные значения. Рассмотрим пример, в котором пользователю требуется ввести верхний и нижний пределы, определяющие диапазон значений. В этом случае в программе наверняка понадобится проверка, не превышает ли первое значение второе (обычно при указании диапазонов предполагается, что первое значение меньше второго). Также может стать необходимой проверка вхождения обоих значений в приемлемые пределы. К примеру, поиск в архиве может не работать со значениями для года, которые меньше 1958 или больше 2014. Такую проверку также имеет смысл реализовать в виде функции.

Рассмотрим одну из возможностей. В следующей функции предполагается, что в код включен заголовочный файл stdbool.h. Если в вашей системе тип _Bool отсутствует, можете подставить тип int для bool, 1 для true и 0 для false. Обратите внимание, что эта функция возвращает true, если ввод является недопустимым; отсюда и ее название bad_limits():

```
bool bad_limits(long begin, long end,
                long low, long high)
{
   bool not_good = false;
   if (begin > end)
   {
      printf("%ld не меньше чем %ld.\n", begin, end);
      not_good = true;
   }
   if (begin < low || end < low)
   {
      printf("Значения должны быть равны %d или больше.\n", low);
      not_good = true;
   }
   if (begin > high || end > high)
   {
      printf("Значения должны быть равны %d или меньше.\n", high);
      not_good = true;
   }
   return not_good;
}
```

В листинге 8.7 эти две функции применяются для предоставления целых чисел арифметической функции, которая вычисляет сумму квадратов всех целых чисел в указанном диапазоне. Программа ограничивает верхние и нижние пределы диапазона значениями 1000 и -1000, соответственно.

Листинг 8.7. Программа checking.c

```
// checking.c -- проверка допустимости ввода
#include <stdio.h>
```

```
#include <stdbool.h>
// проверка, является ли ввод целочисленным
long get_long(void);
// проверка, допустимы ли границы диапазона
bool bad_limits(long begin, long end,
                long low, long high);
// вычисление суммы квадратов целых чисел от a до b
double sum_squares(long a, long b);
int main(void)
{
   const long MIN = -10000000L;    // нижний предел диапазона
   const long MAX = +10000000L;    // верхний предел диапазона
   long start;                     // начало диапазона
   long stop;                      // конец диапазона
   double answer;
   printf("Эта программа вычисляет сумму квадратов "
         "целых чисел в заданном диапазоне.\nНижняя граница не должна "
         "быть меньше -10000000, \nа верхняя не должна быть "
         "больше +10000000.\nвведите значения "
         "пределов (для завершения введите 0 для обоих пределов):\n"
         "нижний предел: ");
   start = get_long();
   printf("верхний предел: ");
   stop = get_long();
   while (start !=0 || stop != 0)
   {
      if (bad_limits(start, stop, MIN, MAX))
         printf("Повторите попытку.\n");
      else
      {
         answer = sum_squares(start, stop);
         printf("Сумма квадратов целых чисел ");
         printf("от %ld до %ld равна %g\n",
         start, stop, answer);
      }
      printf("Введите значения пределов (для завершения "
             "введите 0 для обоих пределов):\n");
      printf("нижний предел: ");
      start = get_long();
      printf("верхний предел: ");
      stop = get_long();
   }
   printf("Программа завершена.\n");
   return 0;
}

long get_long(void)
{
   long input;
   char ch;
   while (scanf("%ld", &input) != 1)
   {
      while ((ch = getchar()) != '\n')
         putchar(ch);  // отбросить неправильный ввод
      printf(" не является целочисленным.\nВведите ");
      printf("целое число, такое как 25, -178 или 3: ");
   }
   return input;
}
```

```
double sum_squares(long a, long b)
{
   double total = 0;
   long i;

   for (i = a; i <= b; i++)
      total += (double)i * (double)i;

   return total;
}

bool bad_limits(long begin, long end,
                long low, long high)
{
   bool not_good = false;

   if (begin > end)
   {
      printf("%ld не меньше чем %ld.\n", begin, end);
      not_good = true;
   }
   if (begin < low || end < low)
   {
      printf("Значения должны быть равны %d или больше.\n", low);
      not_good = true;
   }
   if (begin > high || end > high)
   {
      printf("Значения должны быть равны %d или меньше.\n", high);
      not_good = true;
   }

   return not_good;
}
```

Ниже приведены результаты выполнения этой программы:

```
Эта программа вычисляет сумму квадратов целых чисел в заданном диапазоне.
Нижняя граница не должна быть меньше -10000000,
а верхняя не должна быть больше +10000000.
Введите значения пределов (для завершения введите 0 для обоих пределов):
нижний предел: low
low не является целочисленным.
Введите целое число, такое как 25, -178 или 3: 3
верхний предел: a big number
a big number не является целочисленным.
Введите целое число, такое как 25, -178 или 3: 12
Сумма квадратов целых чисел от 3 до 12 равна 645
Введите значения пределов (для завершения введите 0 для обоих пределов):
нижний предел: 80
верхний предел: 10
80 не меньше 10.
Повторите попытку.
Введите значения пределов (для завершения введите 0 для обоих пределов):
нижняя граница: 0
верхняя граница: 0
Программа завершена.
```

Анализ программы

Вычислительное ядро (функция sum_squares()) программы checking.c совсем короткое, но поддержка проверки допустимости ввода делает его более сложным, чем в рассмотренных ранее примерах. Давайте взглянем на некоторые его элементы, для начала сосредоточившись на общей структуре программы.

Мы следовали модульному подходу, используя отдельные функции (модули) для проверки допустимости ввода и для управления отображением. Чем крупнее программа, тем важнее становится модульный подход.

Функция main() управляет потоком, делегируя задачи другим функциям. Функция get_int() применяется в ней для получения значений, цикл while — для их обработки, функция badlimits() — для проверки допустимости значений и функция sum_squares() — для выполнения действительных вычислений:

```
start = get_long();
printf("верхний предел: ");
stop = get_long();
while (start !=0 || stop != 0)
{
   if (bad_limits(start, stop, MIN, MAX))
      printf("Повторите попытку.\n");
   else
   {
      answer = sum_squares(start, stop);
      printf("Сумма квадратов целых чисел ");
      printf("от %ld до %ld равна %g\n",
      start, stop, answer);
   }
   printf("пределов (для завершения "
         "введите 0 для обоих пределов):\n");
   printf("нижний предел: ");
   start = get_long();
   printf("верхний предел: ");
   stop = get_long();
}
```

Поток ввода и числа

При написании кода для обработки некорректного ввода, такого как в листинге 8.7, вы должны иметь четкое представление о том, как происходит ввод в C. Рассмотрим следующую строку ввода:

```
is 28 12.4
```

Для наших глаз этот ввод выглядит подобным строке символов, за которой следует целое число, а за ним значение с плавающей запятой. Для программы на C он выглядит подобно потоку байтов. Первый байт — это символьный код для буквы i, второй байт — символьный код для буквы s, третий байт — символьный код для пробела, четвертый байт — символьный код для цифры 2 и т.д. Таким образом, если функции get_int() встретится эта строка, начинающаяся нецифрового символа, то приведенный ниже код прочитает и отбросит всю строку целиком, включая числа, которые являются просто еще одной разновидностью символов в строке:

```
while ((ch = getchar()) != '\n')
   putchar(ch);                    // отбросить неправильный ввод
```

Хотя входной поток состоит из символов, функция scanf() может преобразовать их в числовое значение, если вы сообщите ей об этом.

Например, рассмотрим следующий ввод:

```
42
```

Если вы используете функцию scanf() со спецификатором %c, она прочитает только символ 4 и сохранит его в переменной типа char. В случае указания спецификатора %s функция прочитает два символа, символ 4 и символ 2, и сохранит их в символьной строке. Если применить спецификатор %d, то scanf() прочитает те же два символа, но приступит к вычислению соответствующего им целочисленного значения, т.е. $4 \times 10 + 2$, или 42. Затем она сохранит целочисленное двоичное представление этого значения в переменной типа int. При указании спецификатора %f функция scanf() прочитает два символа, вычислит, что они соответствуют числовому значению 42.0, выразит это значение во внутреннем представлении с плавающей запятой и сохранит результат в переменной типа float.

Короче говоря, ввод состоит из символов, но scanf() может преобразовать этот ввод в целочисленное значение или значение с плавающей запятой. Использование спецификатора вроде %d или %f ограничивает типы символов, являющихся приемлемым вводом, но функция getchar() и функция scanf(), в которой указан спецификатор %c, принимают любой символ.

Просмотр меню

Во многих компьютерных программах в качестве части пользовательского интерфейса применяется меню. Меню делают программы более удобными для пользователя, но ставят определенные задачи перед программистами. Давайте посмотрим, в чем они состоят. Меню предлагает пользователю варианты ответа. Вот гипотетический пример:

```
Введите букву, соответствующую выбранному варианту:
с. совет              з. звонок
п. подсчет            в. выход
```

В идеальном случае пользователь вводит одну из предложенных букв, и программа действует согласно выбору. Будучи программистом, вы хотите, чтобы этот процесс протекал как можно более гладко. Первая цель заключается в том, что программа должна работать корректно, если пользователь придерживается инструкций. Вторая цель состоит в том, что программа должна работать устойчиво в ситуациях, когда пользователь нарушает эти инструкции. Вполне ожидаемо, что второй цели достичь гораздо труднее, потому что трудно предугадать все возможные нарушения, с которыми может столкнуться программа.

В современных приложениях вместо ориентированного на командную строку подхода, рассмотренного в приведенных примерах, обычно используются графические интерфейсы (кнопки для щелчка, флажки для отметки, значки для касания), однако общий процесс остается во многом таким же — предоставление пользователю вариантов выбора, обнаружение и действие в соответствии с ответом пользователя, а также защита от возможного неправильного использования. Лежащая в основе этих различных интерфейсов программная структура во многом аналогична. Однако применение графического интерфейса может упростить управление вводом за счет ограничения доступных для выбора вариантов.

Задачи

Давайте рассмотрим задачи, которые необходимо решать в программе с реализованным меню. Программа должна получать ответ от пользователя и на его основе вы-

бирать дальнейший ход действий. Кроме того, программа должна обеспечить способ возврата меню для последующего выбора вариантов. Оператор switch в С является естественным механизмом для выбора действий, т.к. каждый выбор пользователя соответствует конкретной метке case. Оператор while можно использовать для предоставления повторяющегося доступа к меню. С помощью псевдокода процесс можно описать следующим образом:

```
получить вариант
пока не выбран вариант 'в'
   переключиться на нужный вариант и выполнить его
   получить следующий вариант
```

На пути к более гладкому выполнению

Цели гладкости программы (гладкость при обработке как корректного, так и некорректного ввода) вступает в игру, когда вы решаете, каким образом реализовать указанный план. Например, одно из действий, которое вы можете предпринять — это исключить в части "получить вариант" неправильные ответы, чтобы оператору switch передавались только корректные варианты. Это предполагает представление процесса ввода в виде функции, которая возвращает только корректные ответы. В сочетании с циклом while и оператором switch получается следующая структура программы:

```
#include <stdio.h>
char get_choice(void);
void count(void);
int main(void)
{
   int choice;

   while ( (choice = get_choice()) != 'в')
   {
      switch (choice)
      {
      case 'с' : printf("Покупайте дешево, продавайте дорого.\n");
                 break;
      case 'з' : putchar('\a'); /* ANSI */
                 break;
      case 'п' : count();
                 break;
      default  : printf("Ошибка!\n");
                 break;
      }
   }
   return 0;
}
```

Функция get_choice() определена так, что она может возвращать только значения 'с', 'з', 'п' и 'в'. Вы используете ее во многом так же, как getchar(), получая значение и сравнивая его со значением завершения (в данном случае 'в'). Выбираемые варианты в меню специально упрощены, чтобы можно было сосредоточиться на структуре программы; вскоре мы рассмотрим функцию count(). Конструкция default удобна для отладки. Если функции get_choice() не удается ограничить свое возвращаемое значение, как было запланировано, то случай default позволяет узнать, что происходит что-то подозрительное.

Функция get_choice()

Ниже приведен псевдокод одной из возможных структур этой функции:

```
показать варианты
получить отклик
пока отклик не является приемлемым
   выдать приглашение на ввод следующего отклика
   получить отклик
```

Вот простая, но довольно неуклюжая реализация:

```
char get_choice(void)
{
   int ch;

   printf("Введите букву, соответствующую выбранному варианту:\n");
   printf("a. совет          c. звонок\n");
   printf("b. подсчет        q. выход\n");
   ch = getchar();
   while ( (ch < 'a' || ch > 'c') && ch != 'q')
   {
      printf("Выберите с, з, п или в.\n");
      ch = getchar();
   }
   return ch;
}
```

Проблема заключается в том, что при буферизированном вводе каждый символ новой строки, сгенерированный нажатием клавиши <Enter>, трактуется как ошибочный отклик. Чтобы сделать интерфейс программы более гладким, эта функция должна пропускать символы новой строки.

Существует несколько способов сделать это. Один из них предусматривает замену getchar() новой функцией по имени get_first(), которая читает первый символ в строке и отбрасывает все остальные. Преимущество такого метода состоит в том, что он трактует введенную строку, например, спа, как просто символ с, но не рассматривает ее как правильный вариант с, за которым следует еще один правильный вариант в виде буквы п, означающей подсчет. С учетом всего этого, функцию ввода можно переписать следующим образом:

```
char get_choice(void)
{
   int ch;

   printf("Введите букву, соответствующую выбранному варианту:\n");
   printf("с. совет          з. звонок \n");
   printf("п. подсчет        в. выход\n");
   ch = get_first();
   while ( ch != 'с' && ch != 'з' && ch != 'п' && ch != 'в')
   {
      printf("Выберите с, з, п или в.\n");
      ch = getfirst();
   }
   return ch;
}

char get_first(void)
{
   int ch;
```

```
    ch = getchar();          /* читать следующий символ */
    while (getchar() != '\n')
        continue;            /* отбросить оставшуюся часть строки */
    return ch;
}
```

Смешивание символьного и числового ввода

Создание меню является еще одной иллюстрацией того, что смешивание символьного и числового ввода может привести к проблемам. Предположим, например, что функция count() (выбор n) имеет такой вид:

```
void count(void)
{
    int n,i;

    printf("До какого предела вести подсчет? Введите целое число:\n");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
        printf("%d\n", i);
}
```

Если вы ответите вводом 3, функция scanf() прочитает 3 и оставит символ новой строки в качестве следующего символа во входной очереди. Следующий вызов get_choice() привел бы к тому, что функция get_first() возвратила бы этот символ новой строки, что привело бы к нежелательному поведению.

Один из способов устранить эту проблему — переписать функцию get_first() так, чтобы она возвращала следующий непробельный символ, а не просто любой следующий символ. Мы оставляем эту задачу для самостоятельного выполнения. Второй подход — заставить саму функцию count() следить за порядком и удалять символ новой строки. Именно этот подход применяется в следующем примере:

```
void count(void)
{
    int n,i;

    printf("До какого предела вести подсчет? Введите целое число:\n");
    n = get_int();
    for (i = 1; i <= n; i++)
        printf("%d\n", i);
    while ( getchar() != '\n')
        continue;
}
```

В этой функции также используется функция, которая похожа на get_long() из листинга 8.7, но имеет имя get_int() и извлекает значение типа int вместо long; вспомните, что исходная версия функции проверяет допустимость ввода и предоставляет пользователю возможность повторить попытку. В листинге 8.8 показан окончательный вариант программы с меню.

Листинг 8.8. Программа menuette.c

```
/* menuette.c -- технология меню */
#include <stdio.h>
char get_choice(void);
char get_first(void);
int get_int(void);
void count(void);
```

```
int main(void)
{
    int choice;
    void count(void);
    while ( (choice = get_choice()) != 'q')
    {
        switch (choice)
        {
        case 'с' : printf("Покупайте дешево, продавайте дорого.\n");
                   break;
        case 'з' : putchar('\a'); /* ANSI */
                   break;
        case 'п' : count();
                   break;
        default : printf("Ошибка!\n");
                  break;
        }
    }
    printf("Программа завершена.\n");
    return 0;
}
void count(void)
{
    int n,i;
    printf("До какого предела вести подсчет? Введите целое число:\n");
    n = get_int();
    for (i = 1; i <= n; i++)
        printf("%d\n", i);
    while ( getchar() != '\n')
        continue;
}
char get_choice(void)
{
    int ch;
    printf("Введите букву, соответствующую выбранному варианту:\n");
    printf("с. совет          з. звонок \n");
    printf("п. подсчет         в. выход\n");
    ch = get_first();
    while ( ch != 'с' && ch != 'з' && ch != 'п' && ch != 'в')
    {
        printf("Выберите с, з, п или в.\n");
        ch = get_first();
    }
    return ch;
}
char get_first(void)
{
    int ch;
    ch = getchar();
    while (getchar() != '\n')
        continue;
    return ch;
}
int get_int(void)
{
    int input;
    char ch;
    while (scanf("%d", &input) != 1)
    {
        while ((ch = getchar()) != '\n')
            putchar(ch);  // отбросить неправильный ввод
        printf(" не является целочисленным.\nВведите ");
        printf("целое число, такое как 25, -178 или 3: ");
    }
    return input;
}
```

Ниже приведены результаты выполнения этой программы:

```
Введите букву, соответствующую выбранному варианту:
с. совет            з. звонок
п. подсчет          в. выход
с
Покупайте дешево, продавайте дорого.
Введите букву, соответствующую выбранному варианту:
с. совет            з. звонок
п. подсчет          в. выход
подсчет
До какого предела вести подсчет? Введите целое число:
два
два не является целочисленным.
Введите целое число, такое как 25, -178 или 3: 5

1
2
3
4
5
Введите букву, соответствующую выбранному варианту:
с. совет            з. звонок
п. подсчет          в. выход
d
Выберите с, з, п или в.
в
```

Иногда добиться желаемой гладкой работы интерфейса с меню может быть затруднительно, но после разработки жизнеспособного подхода вы сможете применять его в разнообразных ситуациях.

Также обратите внимание на то, как каждая функция, столкнувшись с необходимостью выполнить чуть более сложную задачу, делегирует это другой функции, существенно повышая модульность программы.

Ключевые понятия

Программы на языке C рассматривают ввод как поток входящих байтов. Функция `getchar()` интерпретирует каждый байт как символьный код. Функция `scanf()` воспринимает ввод аналогично, но с помощью спецификаторов преобразования она может преобразовать символьный ввод в числовые значения. Многие операционные системы предлагают механизм перенаправления, который позволяет подставлять файл вместо клавиатуры для ввода и вместо монитора для вывода.

Часто программы ожидают определенной формы ввода. Предугадывая возможные ошибки ввода, и предусматривая средства их обработки в программе, можно значительно увеличить надежность программы и дружественность по отношению к пользователю.

В случае небольшой программы проверка допустимости ввода может оказаться наиболее сложной частью кода. Кроме того, здесь открывается множество путей. Например, когда пользователь вводит некорректную информацию, вы можете завершить программу, предоставить пользователю фиксированное количество попыток для приведения входных данных в надлежащий вид либо предложить неограниченное число таких попыток.

Резюме

Многие программы используют функцию `getchar()` для посимвольного чтения входных данных. Обычно в системах применяется *построчно буферизированный ввод*, означающий, что входные данные передаются в программу, когда нажимается клавиша <Enter>. Нажатие клавиши <Enter> генерирует символ новой строки, которому может понадобиться уделить внимание в коде. Стандарт ANSI C требует применения буферизированного ввода.

Язык C предлагает семейство функций, называемое *стандартным пакетом ввода-вывода*, который позволяет применять унифицированный подход при работе с различными формами файлов в разных системах. Функции `getchar()` и `scanf()` принадлежат этому семейству. Обе они возвращают значение `EOF` (определенное в `stdio.h`), когда обнаруживают конец файла. Системы Unix позволяют эмулировать условие конца файла с клавиатуры путем нажатия <Ctrl+D> в начале строки; в системах DOS для этого используется комбинация <Ctrl+Z>.

Многие операционные системы, включая Unix и DOS, поддерживают средство *перенаправления*, которое позволяет применять для ввода и вывода файлы вместо клавиатуры и экрана. Программы, которые читают ввод до тех пор, пока не встретится `EOF`, могут использоваться либо с клавиатурным вводом и эмулированными сигналами конца файла, либо с перенаправленными файлами.

Смешивание вызовов `scanf()` с вызовами `getcnar()` может приводить к проблемам в случаях, когда `scanf()` оставляет символ новой строки во входных данных непосредственно перед вызовом `getchar()`. Тем не менее, зная о такой проблеме, ее можно обойти программно.

При написании программы тщательно планируйте пользовательский интерфейс. Постарайтесь предусмотреть все виды ошибок, которые могут совершить пользователи, и затем проектируйте программу так, чтобы их обрабатывать.

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля приведены в приложении А.

1. Выражение `putchar(getchar())` является допустимым; что оно делает? Допустимо ли будет выражение `getchar(putchar())`?
2. Какие действия выполняют следующие операторы?
   а. `putchar('H');`
   б. `putchar('\007');`
   в. `putchar('\n');`
   г. `putchar('\b');`
3. Предположим, что имеется исполняемая программа по имени `count`, которая подсчитывает количество символов во входных данных. Напишите команду для среды командной строки, которая использует программу `count` для подсчета количества символов в файле `essay` и для сохранения результата в файле `essayct`.
4. При наличии программы и файлов, описанных в вопросе 3, какие из приведенных ниже команд являются допустимыми?
   а. `essayct <essay`
   б. `count essay`
   в. `essay >count`
5. Что такое `EOF`?

6. Каким будет вывод каждого из показанных далее фрагментов для указанного ввода (предположите, что переменная ch имеет тип int и ввод является буферизированным)?

   а. Ввод выглядит следующим образом:

   ```
   If you quit, I will.[enter]
   ```

   Фрагмент программы имеет вид:

   ```
   while ((ch = getchar()) != 'i')
      putchar(ch);
   ```

   б. Ввод выглядит следующим образом:

   ```
   Harhar[enter]
   ```

   Фрагмент программы имеет вид:

   ```
   while ((ch = getchar()) != '\n')
   {
   putchar(ch++);
      putchar(++ch);
   }
   ```

7. Как в C учитываются разные соглашения относительно файлов и символов новой строки, принятые в различных компьютерных системах?
8. С какой потенциальной проблемой вы столкнетесь при смешивании символьного и числового ввода в системе с буферизированным вводом?

Упражнения по программированию

В некоторых из описанных ниже упражнений требуется прекращение ввода по достижении EOF. Если в вашей операционной системе перенаправление реализовано неудобно или вообще отсутствует, воспользуйтесь какой-то другой проверкой для прекращения ввода, такой как чтение символа &.

1. Напишите программу, которая подсчитывает количество символов во входных данных до достижения конца файла.
2. Напишите программу, которая читает ввод как поток символов, пока не встретит EOF. Программа должна выводить каждый введенный символ и его десятичный код ASCII. Следует отметить, что в кодировке ASCII символы, предшествующие пробелу, являются непечатаемыми. Трактуйте их особым образом. Если непечатаемым символом является символ новой строки или символ табуляции, выводите, соответственно, \n или \t. В противном случае воспользуйтесь нотацией управляющих символов. Например, ASCII-код 1 — это комбинация <Ctrl+A>, которую можно отобразить как ^A. Обратите внимание, что ASCII-код символа A представляет собой значение <Ctrl+A> плюс 64. Аналогичная зависимость имеется и для других непечатаемых символов. Выводите по 10 пар в строке, кроме случая, когда встречается символ новой строки. (На заметку: операционная система может иметь специальные интерпретации для некоторых управляющих символов и не допускать их попадания в программу.)
3. Напишите программу, которая читает ввод как поток символов, пока не встретит EOF. Программа должна сообщать количество прописных букв, количество строчных букв и количество остальных символов во входных данных. Можете предполагать, что числовые значения для строчных букв являются последовательными, и то же самое справедливо для прописных букв. Либо для большей переносимости можете использовать подходящие классификационные функции из библиотеки ctype.h.

4. Напишите программу, которая читает ввод как поток символов, пока не встретит EOF. Программа должна сообщать среднее количество букв в словах. Не считайте пробельные символы в словах буквами. На самом деле, также не должны учитываться и знаки препинания, но в данном упражнении об этом можно не беспокоиться. (Для учета знаков препинания можно воспользоваться функцией `ispunct()` из семейства `ctype.h`.)
5. Модифицируйте программу угадывания чисел из листинга 8.4, чтобы реализовать более интеллектуальную стратегию угадывания. Например, программа может изначально предположить число 50 и запросить, больше ли оно задуманного, меньше его или же это и есть задуманное число. Если, скажем, предположение меньше задуманного числа, следующая догадка должна находиться посредине между 50 и 100, т.е. 75. Если данное предположение больше задуманного числа, то следующая догадка должна располагаться посредине между 75 и 50 и т.д. Используя такую стратегию *двоичного поиска*, программа быстро найдет правильный ответ, во всяком случае, если пользователь не будет обманывать.
6. Модифицируйте функцию `get_first()` из листинга 8.8 так, чтобы она возвращала первый встреченный непробельный символ. Протестируйте ее в какой-нибудь простой программе.
7. Модифицируйте упражнение по программированию 8 из главы 7 так, чтобы пункты меню помечались буквами, а не цифрами; для прекращения ввода используйте букву `q` вместо цифры `5`.
8. Напишите программу, которая выводит на экран меню, предлагающее выбрать сложение, вычитание, умножение или деление. После выбора программа должна запросить два числа и затем выполнить затребованную операцию. Программа должна принимать только варианты, предлагаемые в меню. Для чисел должен использоваться тип `float` и программа должна предоставлять пользователю возможность повторно вводить числа, если он ввел нечисловые данные. В случае деления программа должна предложить пользователю ввести другое значение, если он ввел для второго операнда значение `0`. Выполнение такой программы должно иметь примерно такой вид:

```
Выберите желаемую операцию:
с. сложение         в. вычитание
у. умножение        д. деление
з. завершение
с
Введите первое число: 22.4
Введите второе число: один
один не является числом.
Введите число, такое как 2.5, -1.78E8 или 3: 1
22.4 + 1 = 23.4
Выберите желаемую операцию:
с. сложение         в. вычитание
у. умножение        д. деление
з. завершение
д
Введите первое число: 18.4
Введите второе число: 0
Введите число, отличное от 0: 0.2
18.4 / 0.2 = 92
Выберите желаемую операцию:
с. сложение         в. вычитание
у. умножение        д. деление
з. завершение
з
Программа завершена.
```

9

Функции

В ЭТОЙ ГЛАВЕ...

- Ключевые слова: `return`
- Операции: * (унарная), &(унарная)
- Функции и их определение
- Использование аргументов и возвращаемых значений
- Использование переменных-указателей в качестве аргументов функций
- Типы функций
- Прототипы ANSI C
- Рекурсия

Как вы собираетесь организовать программу? Проектная философия С предусматривает использование функций в качестве строительных блоков. Мы уже полагались на стандартную библиотеку С, когда применяли такие функции, как `printf()`, `scanf()`, `getchar()`, `putchar()` и `strlen()`. Теперь мы готовы перейти к более активным действиям — созданию собственных функций. Некоторые аспекты этого процесса затрагивались в предшествующих главах, а в этой главе вся ранее полученная информация будет объединена и расширена.

Обзор функций

Прежде всего, что собой представляет функция? *Функция* — это самодостаточная единица кода программы, спроектированная для выполнения отдельной задачи. Структура функции и способы ее возможного использования определяются синтаксическими правилами. В языке С функция играет ту же самую роль, которую в других языках программирования играют функции, подпрограммы и процедуры, хотя детали могут отличаться. Некоторые функции приводят к выполнению действия. Например, функция `printf()` выводит данные на экран. Другие функции возвращают значение, которое будет применяться в программе. Например, функция `strlen()` сообщает программе длину указанной строки. В общем случае функция может одновременно выполнять действия и возвращать значения.

Почему вы должны использовать функции? Прежде всего, они избавляют от необходимости в многократном написании одного и того же кода. Если в программе нужно выполнять определенную задачу несколько раз, достаточно однажды написать подходящую функцию. Затем эту функцию можно применять внутри программы там, где она необходима, или же использовать ее в разных программах, подобно тому, как во многих программах была задействована функция `putchar()`. Кроме того, даже если задача решается всего лишь один раз в единственной программе, использование функции имеет смысл, т.к. это делает программу более модульной, таким образом улучшая ее читабельность и упрощая внесение изменений либо исправлений. Для примера предположим, что нужно написать программу, которая выполняет следующие действия:

- читает список чисел;
- сортирует эти числа;
- находит среднее значение этих чисел;
- выводит гистограмму.

Можно было бы написать такую программу:

```
#include <stdio.h>
#define SIZE 50
int main(void)
{
   float list[SIZE];

   readlist(list, SIZE);
   sort(list, SIZE);
   average(list, SIZE);
   bargraph(list, SIZE);

   return 0;
}
```

Конечно, вам придется также написать четыре функции readlist(), sort(), average() и bargraph(), но это уже детали. Описательные имена функций проясняют назначение и организацию программы. Затем над каждой функцией можно работать по отдельности, пока она не начнет успешно справляться со своей задачей, а после того, как вы сделаете эти функции достаточно общими, их можно будет многократно применять в других программах.

Многие программисты предпочитают думать о функции как о "черном ящике", определенном в терминах информации, которая в него поступает (его ввод), и значения или действия, которое он производит (его вывод). Вас не заботит то, что происходит внутри черного ящика, если только вы сами не занимаетесь разработкой этой функции. Например, когда вы используете функцию printf(), то знаете, что должны передать ей управляющую строку и возможно некоторые аргументы. Вам также известен вывод, который функция printf() должна сгенерировать. Однако вы вовсе не задумываетесь о коде, реализующем printf(). Такой подход в отношении функций позволяет сосредоточиться на общей структуре программы, не отвлекаясь на детали. До того, как приступать к написанию кода, тщательно обдумайте, что должна делать функция и какова ее роль в программе.

Что вы должны знать о функциях? Необходимо знать, как их правильно определять, вызывать и обеспечивать взаимодействие между ними. Чтобы освежить эти моменты в памяти, мы начнем с рассмотрения очень простого примера, а затем будем добавлять в него новые возможности, пока не будет получена полная картина.

Создание и использование простой функции

Нашей первой скромной целью является создание функции, которая выводит 40 звездочек в строке. Чтобы придать этой функции смысл, мы включим ее в программу, которая выводит простой заголовок письма. Полный код программы приведен в листинге 9.1. Она состоит из функций main() и starbar().

Листинг 9.1.Программа lethead2.c

```
/* lethead2.c */
#include <stdio.h>
#define NAME "GIGATHINK, INC."
#define ADDRESS "101 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
#define WIDTH 40

void starbar(void);      /* прототип функции */

int main(void)
{
   starbar();
   printf("%s\n", NAME);
   printf("%s\n", ADDRESS);
   printf("%s\n", PLACE);
   starbar();            /* использование функции   */
   return 0;
}

void starbar(void)       /* определение функции     */
{
   int count;
   for (count = 1; count <= WIDTH; count++)
      putchar('*');
   putchar('\n');
}
```

Вывод программы выглядит следующим образом:

```
****************************************
GIGATHINK, INC.
101 Megabuck Plaza
Megapolis, CA 94904
****************************************
```

Анализ программы

Ниже отмечены некоторые важные аспекты этой программы.

- Идентификатор `starbar` применяется в трех отдельных контекстах: в *прототипе функции*, который сообщает компилятору разновидность функции `starbar()`, в *вызове функции*, который приводит к выполнению функции, и в *определении функции*, где в точности указано все, что делает функция.
- Подобно переменным, функции имеют типы. В любой программе, в которой используется функция, предварительно должен быть объявлен тип этой функции. Поэтому данный прототип ANSI C предшествует объявлению функции `main()`:

  ```
  void starbar(void);
  ```

  Круглые скобки указывают, что `starbar` является именем функции. Первое ключевое слово `void` – это тип функции; тип `void` говорит о том, что функция не возвращает значения. Второе слово `void` (в круглых скобках) означает, что функция не принимает аргументов. Точка с запятой указывает на то, что функция объявляется, а не определяется. То есть эта строка сообщает о том, что в программе применяется функция `starbar()`, что эта функция не возвращает значения и не принимает аргументов и что компилятор должен искать ее определение где-то в другом месте. Для компиляторов, не распознающих прототипы ANSI C, просто объявите тип так:

  ```
  void starbar();
  ```

  Следует отметить, что некоторые очень старые компиляторы не распознают тип `void`. В этом случае для функций, которые не возвращают значения, используйте тип `int`. И постарайтесь подыскать более новый компилятор.
- В общем случае прототип указывает как тип возвращаемого функцией значения, так и типы ожидаемых ею аргументов. Обобщенно эту информацию называют *сигнатурой* функции. В данном конкретном случае сигнатура указывает, что функция не имеет ни возвращаемого значения, ни аргументов.
- Прототип функции `starbar()` размещен в программе перед функцией `main()`; вместо этого его можно было бы поместить внутрь функции `main()` там, где находятся объявления переменных. Допустим любой из этих способов.
- Программа вызывает функцию (обращается к функции) `starbar()` из функции `main()`, для чего указывается ее имя, круглые скобки и точка с запятой, таким образом создавая оператор:

  ```
  starbar();
  ```

  Это форма вызова функции типа `void`. Каждый раз, когда управление сталкивается с оператором `starbar();`, оно ищет функцию `starbar()` и выполняет содержащиеся в ней инструкции. По завершении выполнения кода `starbar()` управление возвращается на следующую строку в *вызывающей функции* – в данном случае `main()` (рис. 9.1). (Точнее говоря, компилятор преобразует программу на языке C в код на машинном языке, который ведет себя описанным образом.)

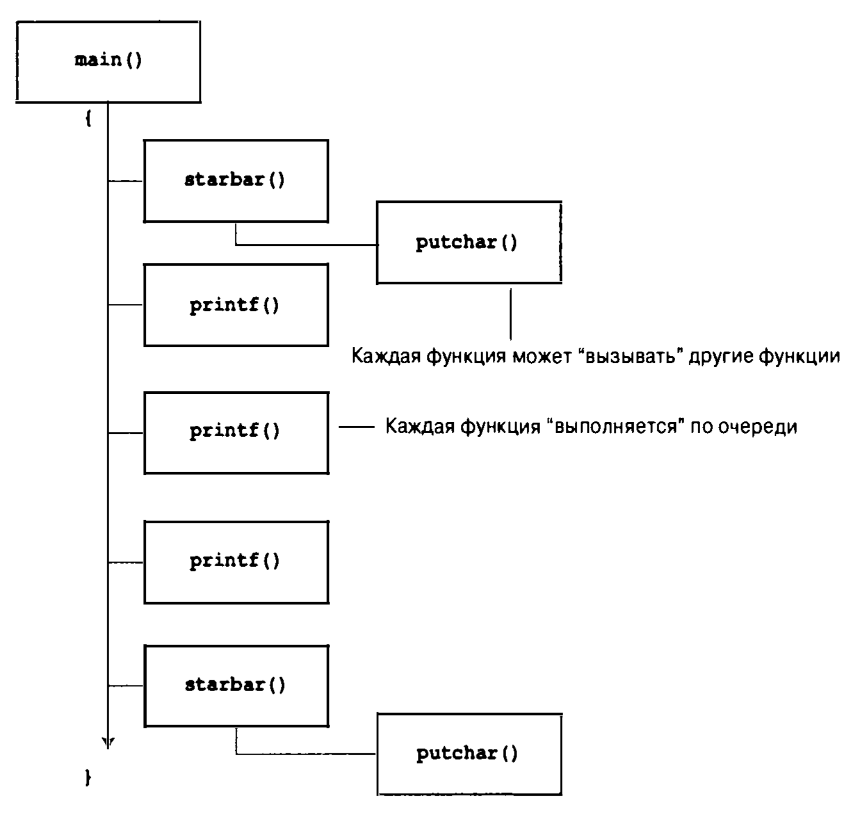


Рис. 9.1. Поток управления в программе `lethead1.c` *(листинг 9.1)*

- При определении функции `starbar()` в программе применяется та же форма, что и при определении `main()`. Определение начинается с типа, имени и круглых скобок. Далее следует открывающая фигурная скобка, объявление используемых переменных, операторы функции и закрывающая фигурная скобка (рис. 9.2). Обратите внимание, что после этого экземпляра функции `starbar()` точка с запятой не указана. Отсутствие точки с запятой говорит компилятору о том, что функция `starbar()` определяется, а не вызывается или прототипируется.
- Функции `starbar()` и `main()` в программе находятся в одном файле. Их можно также разнести по двум отдельным файлам. Форму с одним файлом легче компилировать. Два отдельных файла упрощают применение одной функции в разных программах. Если вы помещаете функцию в отдельный файл, то должны поместить в него также необходимые директивы `#define` и `#include`. Использование двух и более файлов обсуждается позже, а пока все функции мы будем хранить вместе в одном файле. Закрывающая фигурная скобка `main()` указывает компилятору, где эта функция заканчивается, а следующий за ней заголовок `starbar()` сообщает компилятору о том, что `starbar()` является функцией.
- Переменная `count` в функции `starbar()` является *локальной*. Это означает, что она известна только этой функции. Имя `count` можно применять в других функциях, включая `main()`, и это не приведет к конфликту. Просто в программе будут существовать отдельные независимые друг от друга переменные, имеющие одинаковые имена.

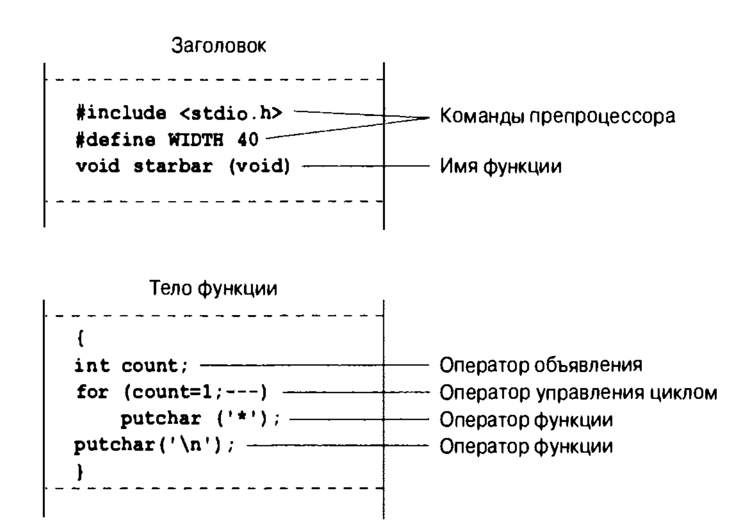


Рис. 9.2. Структура простой функции

Если думать о функции starbar() как о черном ящике, то ее действие заключается в выводе строки звездочек. Она не принимает входные данные, т.к. ей не нужна какая-либо информация от вызывающей функции. Она не предоставляет (т.е. не *возвращает*) информацию функции main(), поэтому starbar() не имеет возвращаемого значения. Короче говоря, функция starbar() не нуждается в каком-либо обмене данными с вызывающей функцией.

Давайте создадим функцию, для которой такой обмен данным необходим.

Аргументы функции

Показанный ранее заголовок письма выглядел бы намного лучше, если бы текст располагался по центру. Текст можно центрировать, поместив подходящее количество ведущих пробелов перед выводом собственно текста. Такое поведение аналогично функции starbar(), которая выводила заданное число звездочек, но теперь необходимо выводить определенное количество пробелов. Вместо написания отдельной функции для каждой задачи мы создадим одну, но более универсальную функцию, которая решает обе задачи. Назовем эту новую функцию show_n_char() (имя означает, что символ отображается *n* раз). Единственное изменение касается того, что вместо использования встроенных значений для отображаемого символа и количества повторений в функции show_n_char() для этого будут применяться аргументы.

Давайте перейдем к деталям. Предположим, что доступное пространство имеет ширину 40 символов. Строка из звездочек содержит 40 символов, в точности соответствуя по ширине, и вызов show_n_char('*', 40) должен выводить эту строку точно так же, как это ранее делала функция starbar(). Что можно сказать о пробелах, используемых для центрирования строки GIGATHINK, INC.? Строка GIGATHINK, INC. имеет ширину 15 символов, поэтому в первой версии программы за заголовком следовали 25 пробелов. Для центрирования строки необходимо начать строку с 12 пробелов, что даст в результате 12 пробелов с одной стороны и 13 пробелов с другой. Таким образом, можно применять вызов show_n_char(' ', 12).

Помимо аргументов функция show_n_char() будет довольно похожа на starbar(). Одно отличие заключается в том, что show_n_char() не добавляет символ новой строки, как это делает starbar(), поскольку в той же строке может понадобиться вы-

вести другой текст. Переделанная версия программы показана в листинге 9.2. Чтобы продемонстрировать работу аргументов, в программе используются различные их формы.

Листинг 9.2. Программа lethead2.c

```
/* lethead2.c */
#include <stdio.h>
#include <string.h>  /* для strlen() */
#define NAME "GIGATHINK, INC."
#define ADDRESS "101 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
#define WIDTH 40
#define SPACE ' '

void show_n_char(char ch, int num);
int main(void)
{
   int spaces;
   show_n_char('*', WIDTH);      /* использование констант в качестве аргументов */
   putchar('\n');
   show_n_char(SPACE, 12);       /* использование констант в качестве аргументов */
   printf("%s\n", NAME);
   spaces = (WIDTH - strlen(ADDRESS)) / 2;
                                 /* позволить программе вычислить,             */
                                 /* сколько пробелов нужно вывести              */
   show_n_char(SPACE, spaces);  /* использование переменной в качестве аргумента */
   printf("%s\n", ADDRESS);
   show_n_char(SPACE, (WIDTH - strlen(PLACE)) / 2);
                                 /* использование выражения в качестве аргумента  */
   printf("%s\n", PLACE);
   show_n_char('*', WIDTH);
   putchar('\n');
   return 0;
}
/* определение функции show_n_char() */
void show_n_char(char ch, int num)
{
   int count;

   for (count = 1; count <= num; count++)
      putchar(ch);
}
```

Вот результат выполнения программы:

```
****************************************
            GIGATHINK, INC.
           101 Megabuck Plaza
          Megapolis, CA 94904
****************************************
```

Теперь давайте посмотрим, как построить функцию, которая принимает аргументы. После этого мы взглянем, как ею пользоваться.

Определение функции с аргументами: формальные параметры

Определение функции начинается со следующего заголовка ANSI C:

```
void show_n_char(char ch, int num)
```

Эта строка информирует компилятор о том, что функция show_n_char() принимает два аргумента с именами ch и num, ch имеет тип char, а num — тип int. Переменные ch и num называются *формальными аргументами* или (что более предпочтительно в настоящее время) *формальными параметрами*. Подобно переменным, определенным внутри функции, формальные параметры являются локальными переменными, закрытыми для функции. Это означает, что можно не беспокоиться, если их имена будут дублироваться в других функциях. Значения этим переменным будут присваиваться при вызове функции. Обратите внимание, что форма ANSI C требует, чтобы каждой переменной предшествовал ее тип. То есть, в отличие от обычных объявлений, нельзя применять список переменных, которые имеют один и тот же тип:

```
void dibs(int x, y, z)          /* некорректный заголовок функции */
void dubs(int x, int y, int z)  /* правильный заголовок функции   */
```

В стандарте ANSI C распознается также форма, которая использовалась до появления ANSI C, но она характеризуется как устаревшая и выходящая из употребления:

```
void show_n_char(ch, num)
char ch;
int num;
```

Здесь в круглых скобках содержится список имен аргументов, однако их типы объявляются позже. Обратите внимание, что аргументы объявляются перед фигурной скобкой, отмечающей начало тела функции, тогда как обычные локальные переменные объявляются после фигурной скобки. Такая форма позволяет применять списки имен переменных, разделенных запятыми, если эти переменные имеют один и тот же тип:

```
void dibs(x, y, z)
int x, y, z;                    /* допустимо */
```

Стандарт ANSI C направлен на то, чтобы постепенно вывести из употребления форму, применяемую ранее. Вы должны быть осведомлены об этой форме, чтобы понимать старый код, но в новых программах необходимо использовать современную форму. (Стандарты C99 и C11 продолжают предупреждать о грядущем ее устаревании.)

Несмотря на то что функция show_n_char() принимает значения из main(), она ничего не возвращает, поэтому show_n_char() имеет тип void.

Теперь посмотрим, как пользоваться этой функцией.

Создание прототипа функции с аргументами

Мы применяем прототип ANSI C, чтобы объявить функцию перед ее применением:

```
void show_n_char(char ch, int num);
```

Когда функция принимает аргументы, прототип отражает их количество и типы, используя разделенный запятыми список типов. При желании имена переменных в прототипе можно не указывать:

```
void show_n_char(char, int);
```

Применение имен переменных в прототипе не приводит к действительному созданию этих переменных. Это просто проясняет тот факт, что char означает переменную типа char и т.д.

Снова напоминаем, что стандарт ANSI C также распознает старую форму объявления функции без списка аргументов:

```
void show_n_char();
```

Со временем эта форма будет исключена из стандарта. Но даже если это не произойдет, формат с прототипом является намного лучшим проектным решением, как будет показано позже. Основная причина, по которой следует знать устаревшую форму, связана с необходимостью чтения ранее написанного кода.

Вызов функции с аргументами: фактические аргументы

Значения ch и num присваиваются с использованием *фактических аргументов* в вызове функции. Рассмотрим первый случай применения show_n_char():

```
show_n_char(SPACE, 12);
```

Фактическими аргументами являются SPACE и 12. Эти значения присваиваются соответствующим формальным параметрам функции show_n_char() — переменным ch и num. Выражаясь кратко, формальный параметр — это переменная в вызванной функции, а фактический аргумент — это конкретное значение, которое вызывающая функция присваивает переменной внутри вызванной функции. Как показывает пример, фактическим аргументом может быть константа, переменная или даже более сложное выражение. Независимо от того, чем является фактический аргумент, он вычисляется, и его значение копируется в соответствующий формальный параметр для функции. Например, рассмотрим финальное использование show_n_char():

```
show_n_char(SPACE, (WIDTH - strlen(PLACE)) / 2);
```

Вычисление длинного выражения, образующего второй фактический аргумент, дает в результате 10. Затем значение 10 присваивается переменной num. Функция не знает, да и не заботится о том, откуда поступает это число — из константы, переменной или более общего выражения. Еще раз подчеркнем, что фактический аргумент представляет собой конкретное значение, которое присваивается переменной, известной как формальный параметр (рис. 9.3). Поскольку вызванная функция работает с данными, скопированными из вызывающей функции, исходные данные в вызывающей функции защищены от любых манипуляций, которые вызванная функция применяет к их копиям.

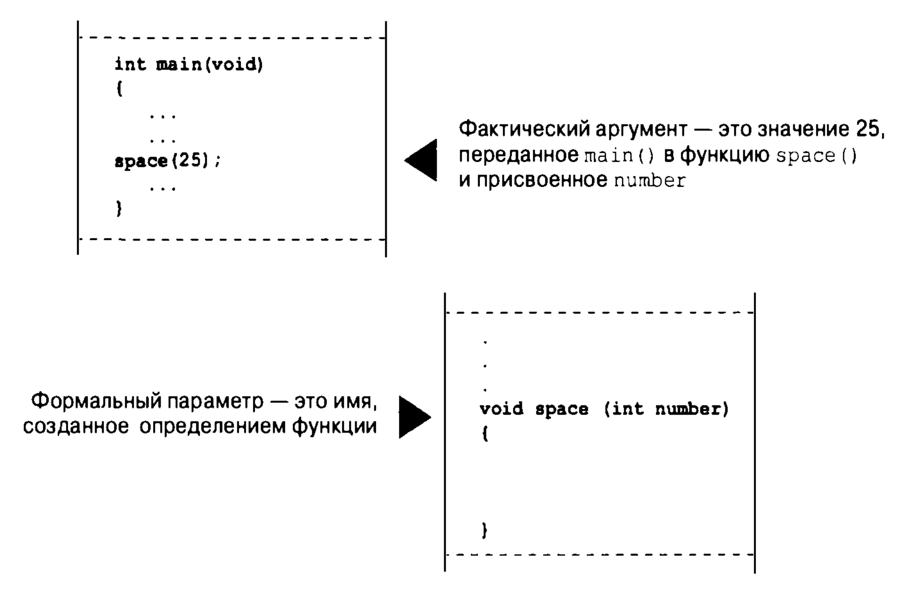


Рис. 9.3. Формальные параметры и фактические аргументы

НА ЗАМЕТКУ! Фактические аргументы и формальные параметры

Фактический аргумент — это выражение, указанное в круглых скобках при вызове функции. Формальный параметр — это переменная, объявленная в заголовке определения функции. Когда функция вызывается, переменные, объявленные как формальные параметры, создаются и инициализируются значениями, которые получены в результате вычисления фактических аргументов. В листинге 9.2 выражения '*' и WIDTH являются фактическими аргументами, когда функция show_n_char() вызывалась первый раз, а SPACE и 11 — фактическими аргументами при втором ее вызове. В определении функции переменные ch и num представляют собой формальные параметры.

Представление в виде черного ящика

При представлении функции show_n_char() в виде черного ящика входными данными являются отображаемый символ и количество пробелов, которые нужно пропустить. Результирующим действием будет вывод символа указанное число раз. Входные данные передаются функции посредством аргументов. Этой информации вполне достаточно для понимания того, как эта функция используется в main(). Кроме того, эта информация служит проектной спецификацией для написания функции.

Тот факт, что ch, num и count – локальные переменные, закрытые в рамках функции show_n_char(), является существенным аспектом подхода с черным ящиком. Если бы в функции main() применялись переменные с такими же именами, то это были бы другие, независимые переменные. То есть, если бы в main() имелась переменная count, то изменение ее значения не привело бы к изменению значения count в show_n_char(), и наоборот. Все, что происходит внутри черного ящика, скрыто от вызывающей функции.

Возврат значения из функции с помощью return

Вы уже видели, как передавать информацию из вызываемой функции в вызванную. Для отправки информации в противоположном направлении используется возвращаемое значение. Чтобы напомнить, как это работает, мы реализуем функцию, которая возвращает меньшее значение из двух аргументов. Назовем эту функцию imin(), т.к. она предназначена для поддержки значений типа int. Кроме того, мы создадим простую функцию main(), единственной целью которой будет проверка работоспособности imin(). Программу, разработанную для такого тестирования функций, иногда называют *драйвером*. Драйвер получает функцию для проверки. Если функция проходит проверку успешно, ее можно применять в более примечательной программе. В листинге 9.3 показан код драйвера и функции выбора минимального значения.

Листинг 9.3. Программа lesser.c

```
/* lesser.c -- выбирает меньшее из двух зол */
#include <stdio.h>
int imin(int, int);

int main(void)
{
   int evil1, evil2;
   printf("Введите два целых числа (или q для завершения):\n");
   while (scanf("%d %d", &evil1, &evil2) == 2)
   {
      printf("Меньшим из двух чисел %d и %d является %d.\n",
         evil1, evil2, imin(evil1,evil2));
      printf("Введите два целых числа (или q для завершения):\n");
   }
```

```
   printf("Программа завершена.\n");
   return 0;
}
int imin(int n,int m)
{
   int min;
   if (n < m)
      min = n;
   else
      min = m;
   return min;
}
```

Вспомните, что функция scanf() возвращает количество успешно прочитанных элементов, поэтому ввод чего-либо, отличного от двух целых чисел, приведет к прекращению выполнения цикла. Ниже приведены результаты запуска этой программы:

```
Введите два целых числа (или q для завершения):
509 333
Меньшим из двух чисел 509 и 333 является 333.
Введите два целых числа (или q для завершения):
-9393 6
Меньшим из двух чисел -9393 и 6 является -9393.
Введите два целых числа (или q для завершения):
q
Программа завершена.
```

Ключевое слово return приводит к тому, что следующее за ним выражение становится возвращаемым значением функции. В данном случае функция возвращает значение, которое было присвоено переменной min. Поскольку min имеет тип int, функция imin() также относится к этому типу.

Переменная min является закрытой для imin(), но с помощью return значение min передается обратно вызывающей функции. Действием показанного ниже оператора будет присваивание значения min переменной lesser:

```
lesser = imin(n,m);
```

Можно ли было бы взамен написать следующий код?

```
imin(n,m);
lesser = min;
```

Нет, потому что вызывающая функция даже не подозревает о существовании переменной min. Вспомните, что переменные внутри функции imin() являются локальными по отношению к ней. Вызов функции imin(evil1,evil2) копирует значения из одного набора переменных в другой.

Возвращаемое значение не только может быть присвоено переменной, но также использоваться как часть выражения. Например, можно написать следующий код:

```
answer = 2 * imin(z, zstar) + 25;
printf("%d\n", imin(-32 + answer, LIMIT));
```

Возвращаемое значение может быть предоставлено любым выражением, а не только переменной. Например, код функции можно сократить до такого вида:

```
/* функция, определяющая минимальное значение, вторая версия */
imin(int n,int m)
{
   return (n < m) ? n : m;
}
```

Результатом вычисления этого условного выражения будет меньшее из значений n и m, после чего данное значение возвращается в вызывающую функцию. Если вы предпочитаете для ясности или ради соблюдения стиля заключать возвращаемое значение в круглые скобки, можете поступать так, хотя круглые скобки здесь не обязательны.

А что, если функция возвращает тип, отличающийся от объявленного?

```
int what_if(int n)
{
   double z = 100.0 / (double) n;
   return z;                    // что произойдет?
}
```

Тогда фактическим значением будет то, что вы получили бы, присвоив указанное возвращаемое значение переменной с объявленным возвращаемым типом. Таким образом, в данном примере конечный итог будет тем же, как если бы вы присвоили значение z переменной типа int и затем возвратили это значение. Предположим, что имеется следующий вызов функции:

```
result = what_if(64);
```

Переменная z получает значение 1.5625. Тем не менее, оператор return возвращает значение 1 типа int.

Применение оператора return дает еще один результат. Он завершает функцию и возвращает управление следующему оператору внутри вызывающей функции. Это происходит даже в случае, если оператор return в функции не является последним. Следовательно, код функции imin() можно записать так, как показано ниже:

```
/* функция, определяющая минимальное значение, третья версия */
imin(int n,int m)
{
   if (n < m)
      return n;
   else
      return m;
}
```

Многие (хотя и не все) практикующие программисты на C считают, что оператор return лучше использовать только один раз и в конце функции, чтобы упростить отслеживание потока управления через код функции. Тем не менее, нет никаких препятствий к применению более одного оператора return даже в такой короткой функции, как показанная выше. В любом случае для пользователя все три версии одинаковы, т.к. все они принимают те же входные данные и генерируют один и тот же вывод. В них отличается только внутренняя реализация. Даже следующая версия работает идентично:

```
/* функция, определяющая минимальное значение, четвертая версия */
imin(int n, int m)
{
   if (n < m)
      return n;
   else
      return m;
   printf("Профессор Флеппард - тщеславный, манерный и тупой щеголь.\n");
}
```

Операторы return предотвращают достижение оператора printf(). Профессор Флеппард может пользоваться скомпилированной версией этой функции в собствен-

ных программах, так никогда и не узнав истинного к нему отношения со стороны своего студента.

Можно также применять оператор такого вида:

```
return;
```

Он приводит к завершению функции и возвращает управление вызывающей функции. Из-за того, что выражение за `return` отсутствует, значение не возвращается, и эта форма должна использоваться только в функциях типа `void`.

Типы функций

Функции должны объявляться с указанием типов. Функция с возвращаемым значением должна быть объявлена с тем же типом, что и у возвращаемого значения. Функция без возвращаемого значения должна быть объявлена с типом `void`. Если функции не назначен тип, то в более ранних версиях С предполагалось, что такая функция относится к типу `int`. Это соглашение восходит к давним временам существования языка С, когда большинство функций имели тип `int`. Однако поддержка такого неявного предположения о типе `int` из стандарта С99 изъята.

Объявление типа является частью определения функции. Имейте в виду, что оно относится к возвращаемому значению, но не к аргументам функции. Например, приведенный ниже заголовок функции указывает на то, что вы определяете функцию, которая принимает два аргумента типа `int`, но возвращает значение типа `double`:

```
double klink(int a, int b)
```

Для корректного использования функции программа должна знать тип функции до первого ее вызова. Один из способов достижения этого предусматривает размещение полного определения функции до ее первого применения. Однако такой метод может затруднить восприятие программы. Кроме того, функции могут быть частью библиотеки С или находиться в каком-то другом файле. Таким образом, обычно вы информируете компилятор о функциях, объявляя их заранее. Например, функция `main()` из листинга 9.3 содержит следующие строки:

```
#include <stdio.h>
int imin(int, int);
int main(void)
{
    int evil1, evil2, lesser;
```

Вторая строка кода устанавливает, что `imin` является именем функции, которая имеет два параметра типа `int` и возвращает значение типа `int`. Теперь компилятор будет знать, каким образом трактовать конструкцию `imin()`, когда она позже встретится в программе.

Мы разместили предварительные объявления функций за пределами функции, в которой они используются. Их также можно поместить внутрь этой функции. Например, начало программы `lesser.c` можно переписать так:

```
#include <stdio.h>
int main(void)
{
    int imin(int, int);      /* объявление функции imin()*/
    int evil1, evil2, lesser;
```

В любом случае основная задача заключается в том, чтобы обеспечить появление объявления функции до ее фактического применения.

В стандартной библиотеке ANSI C функции сгруппированы в семейства, каждое из которых имеет собственный заголовочный файл. Такие заголовочные файлы содержат среди прочего объявления функций в семействах. Например, заголовочный файл stdio.h включает объявления для стандартных библиотечных функций ввода-вывода, таких как printf() и scanf(). Заголовочный файл math.h содержит объявления для множества математических функций, например, объявление

```
double sqrt(double);
```

которое сообщает компилятору о том, что функция sqrt() принимает параметр типа double и возвращает значение типа double. Не путайте эти объявления с определениями. Объявление функции информирует компилятор о том, какой тип имеет функция, а определение функции предоставляет действительный код. Включение заголовочного файла math.h уведомляет компилятор о том, что возвращаемым типом sqrt() является double, но код для функции sqrt() находится в отдельном файле библиотечных функций.

Создание прототипов функций в ANSI C

Традиционная схема объявления функций, используемая до появления ANSI C, обладала тем недостатком, что предусматривала указание типа для возвращаемого значения функции, но не для ее аргументов. Давайте посмотрим, проблемы какого вида возникают в случае объявления функций по старой форме.

Следующее объявление, применяемое до выхода ANSI C, информирует компилятор о том, что функция imin() возвращает значение типа int:

```
int imin();
```

Однако оно ничего не говорит о количестве или типах аргументов этой функции. В результате, если функция imin() используется с некорректным количеством аргументов либо неподходящими их типами, то компилятор не сможет выявить ошибку.

Суть проблемы

Давайте рассмотрим несколько примеров, в которых задействована функция imax(), тесно связанная с imin(). В листинге 9.4 показана программа, в которой функция imax() объявляется устаревшим способом, а затем некорректно применяется.

Листинг 9.4. Программа misuse.c

```
/* misuse.c -- неправильное использование функции */
#include <stdio.h>
int imax();    /* объявление в старом стиле */

int main(void)
{
   printf("Наибольшим значением из %d и %d является %d.\n",
         3, 5, imax(3));
   printf("Наибольшим значением из %d и %d является %d.\n",
         3, 5, imax(3.0, 5.0));
   return 0;
}
int imax(n, m)
int n, m;
{
   return (n > m ? n : m);
}
```

В первом вызове `printf()` не указан один аргумент функции `imax()`, а во втором вызове используются аргументы с плавающей запятой вместо целочисленных. Несмотря на эти ошибки, программа компилируется и запускается.

Вот пример вывода в случае применения XCode 4.6:

```
Наибольшим значением из 3 и 5 является 1606416656.
Наибольшим значением из 3 и 5 является 3886.
```

Запуск программы после компиляции с помощью gcc генерирует значения 1 359 379 472 и 1 359 377 160. Оба эти компилятора работают нормально, просто они становятся жертвами некорректного применения прототипов функций в программе.

Что происходит? Механизмы могут отличаться между системами, но вот что происходит в компьютере IBM PC или VAX. Вызывающая функция помещает аргументы во временную память, называемую *стеком*, и вызванная функция читает их оттуда. Оба эти процесса *не* скоординированы друг с другом. Вызывающая функция решает, какие типы передавать, на основе фактических аргументов в вызове, а вызванная функция читает значения, основываясь на типах своих формальных параметров. Таким образом, вызов `imax(3)` помещает в стек *одно* целочисленное значение. Когда функция `imax()` запускается, она читает из стека *два* целочисленных значений. Но в стек было помещено только одно значение, поэтому вторым прочитанным значением будет то, что случайно находилось в стеке в этот момент.

При втором использовании функции `imax()` ей передаются значения типа `float`. В результате в стек помещаются два значения типа `double`. (Вспомните, что при передаче в качестве аргумента тип `float` повышается до `double`.) В нашей системе это два 64-битных значения, так что в стек попадают 128 битов данных. Когда функция `imax()` читает из стека два значения типа `int`, она извлекает первые 64 бита, т.к. в нашей системе тип `int` занимает 32 бита. По случайному совпадению эти биты соответствовали двум целочисленным значениям, большим из которых оказалось 3 886.

Решение стандарта ANSI C

Подход к решению проблем с несоответствием аргументов, реализованный в стандарте ANSI C, предусматривает разрешение указывать в объявлении функции также и типы переменных. Результатом является *прототип функции* — объявление, в котором устанавливается возвращаемый тип, количество аргументов, а также их типы. Чтобы указать, что функция `imax()` требует два аргумента `int`, ее можно объявить с помощью одного из следующих прототипов:

```
int imax(int, int);
int imax(int a, int b);
```

В первой форме применяется список типов, разделенных запятыми. Во второй форме к типам добавлены имена переменных. Помните, что имена переменных являются фиктивными и не обязаны соответствовать именам, используемым в определении функции.

Располагая этой информацией, компилятор может проверить, совпадает ли вызов функции с прототипом. Указано ли правильное количество аргументов? Имеют ли они корректные типы? В случае несовпадения типов, когда оба типа являются числовыми, компилятор преобразует значения фактических аргументов в типы формальных параметров. Например, вызов `imax(3.0, 5.0)` становится `imax(3, 5)`. Мы модифицировали код в листинге 9.4 для применения прототипа функции. Результат представлен в листинге 9.5.

Листинг 9.5. Программа proto.c

```
/* proto.c -- использует прототипы функции */
#include <stdio.h>
int imax(int, int);      /* прототип */
int main(void)
{
   printf("Наибольшим значением из %d и %d является %d.\n",
          3, 5, imax(3));
   printf("Наибольшим значением из %d и %d является %d.\n",
          3, 5, imax(3.0, 5.0));
   return 0;
}
int imax(int n, int m)
{
   return (n > m ? n : m);
}
```

Попытка компиляции программы из листинга 9.5 приводит к выдаче компилятором сообщения об ошибке, указывающего на то, что вызов imax() содержит слишком мало параметров.

А что можно сказать об ошибках, связанных типами? Для их исследования мы заменили imax(3) вызовом imax(3, 5) и попробовали скомпилировать программу еще раз. На этот раз сообщения об ошибках не было, и мы запустили программу на выполнение. Ниже показан результирующий вывод:

```
Наибольшим значением из 3 и 5 является 5.
Наибольшим значением из 3 и 5 является 5.
```

Как и ожидалось, 3.0 и 5.0 во втором вызове были преобразованы в 3 и 5, чтобы функция могла должным образом обработать входные данные.

Хотя сообщения об ошибке отсутствовали, компилятор выдал предупреждение о том, что тип double был преобразован в тип int и возможна потеря данных. Например, вызов

```
imax(3.9, 5.4)
```

становится эквивалентным следующему вызову:

```
imax(3, 5)
```

Разница между сообщением об ошибке и предупреждением заключается в том, что ошибка предотвращает компиляции, а предупреждение – нет. Некоторые компиляторы выполняют такое приведение типа, не информируя вас. Причина связана с тем, что стандарт не требует вывода предупреждений. Однако многие компиляторы позволяют выбирать уровень выдачи предупреждений, который управляет многословностью компилятора при сообщении о предупреждениях.

Отсутствие аргументов и неопределенные аргументы

Предположим, что вы создали прототип следующего вида:

```
void print_name();
```

Компилятор ANSI C предположит, что вы решили воспользоваться стилем объявления, предшествующим прототипированию, и не будет проверять аргументы. Для отражения того, что функция не принимает аргументов, укажите в круглых скобках ключевое слово void:

```
void print_name(void);
```

Компилятор ANSI C интерпретирует предыдущее выражение как то, что функция `print_name()` не имеет аргументов. Затем он проверяет, не применяете ли вы аргументы при вызове этой функции.

Некоторые функции, такие как `printf()` и `scanf()`, принимают переменное количество аргументов. Например, в `printf()` первым аргументом является строка, но остальные аргументы не фиксированы ни по типам, ни по количеству. Для таких случаев стандарт ANSI C разрешает частичное прототипирование. Например, для `printf()` можно было бы использовать такой прототип:

```
int printf(const char *, ...);
```

Этот прототип указывает, что первым аргументом является строка (данный аспект объясняется в главе 11) и что могут быть указаны дальнейшие аргументы с неопределенной природой.

Заголовочный файл `stdarg.h` в библиотеке функций C предоставляет стандартный способ для определения функции с переменным количеством параметров; детали будут раскрыты в главе 16.

Преимущество прототипов

Прототипы являются мощным дополнением к языку. Они позволяют компилятору выявлять многие ошибки или оплошности, которые могли быть допущены при использовании функций. Если их не обнаружить своевременно, они превратятся в проблемы, которые могут оказаться трудными для отслеживания. Обязаны ли вы применять прототипы? Нет, вы вполне можете использовать старый метод объявления функций (без указания параметров), но никакими преимуществами он не обладает, взамен имея множество недостатков.

Существует один способ избежать прототипа, одновременно сохранив преимущества прототипирования. Причиной указания прототипа является сообщение компилятору о том, как должна использоваться функция, до достижения им первого фактического случая ее применения. Того же результата можно добиться, поместив полное определение функции до ее первого использования. В этом случае определение действует как собственный прототип. Чаще всего это делается с короткими функциями:

```
// следующий код является определением и прототипом
int imax(int a, int b) { return a > b ? a : b; }

int main()
{
   int x, z;
   ...
   z = imax(x, 50);
   ...
}
```

Рекурсия

В языке C функции разрешено вызывать саму себя. Этот процесс называется *рекурсией*. Временами рекурсия бывает сложной, но иногда и очень удобной. Сложность связана с доведением рекурсии до конца, т.к. функция, которая вызывает сама себя, имеет тенденцию делать это бесконечно, если в коде не предусмотрена проверка условия завершения рекурсии.

Рекурсия часто может применяться там, где используется цикл. Иногда более очевидным является решение с циклом, а иногда – решение с рекурсией. Рекурсивные решения более элегантны, но менее эффективны, чем решения с циклами.

Рекурсия в действии

Давайте взглянем на пример рекурсии. Функция main() в листинге 9.6 вызывает функцию up_and_down(). Назовем это "первым уровнем рекурсии". Затем функция up_and_down() вызывает саму себя; назовем это "вторым уровнем рекурсии". Второй уровень вызывает третий уровень рекурсии и т.д. В этом примере настроены четыре уровня рекурсии. Чтобы посмотреть, что происходит внутри, программа не только отображает значение переменной n, но также и значение &n, которое представляет собой адрес ячейки памяти, где хранится переменная n. (Операция & более подробно обсуждается позже в главе. Для вывода адресов в printf() применяется спецификатор %p. Если ваша система не поддерживает этот формат, попробуйте воспользоваться спецификатором %u или %lu.)

Листинг 9.6. Программа recur.c

```
/* recur.c -- иллюстрация рекурсии */
#include <stdio.h>
void up_and_down(int);

int main(void)
{
    up_and_down(1);
    return 0;
}

void up_and_down(int n)
{
    printf("Уровень %d: ячейка n %p\n", n, &n);  // 1
    if (n < 4)
        up_and_down(n+1);
    printf("УРОВЕНЬ %d: ячейка n %p\n", n, &n);  // 2
}
```

Вывод на одной из систем выглядит следующим образом:

```
Уровень 1: ячейка n 0x0012ff48
Уровень 2: ячейка n 0x0012ff3c
Уровень 3: ячейка n 0x0012ff30
Уровень 4: ячейка n 0x0012ff24
УРОВЕНЬ 4: ячейка n 0x0012ff24
УРОВЕНЬ 3: ячейка n 0x0012ff30
УРОВЕНЬ 2: ячейка n 0x0012ff3c
УРОВЕНЬ 1: ячейка n 0x0012ff48
```

Давайте пройдемся по данной программе, чтобы посмотреть, как работает рекурсия. Сначала main() вызывает up_and_down() с аргументом 1. В результате формальный параметр n функции up_and_down() получает значение 1, поэтому первый оператор вывода отображает строку Уровень 1:. Далее, поскольку n меньше 4, функция up_and_down() (уровень 1) вызывает функцию up_and_down() (уровень 2) с фактическим аргументом n + 1, или 2. Это приводит к тому, что n в вызове уровня 2 присваивается значение 2, так что первый оператор вывода отображает строку Уровень 2:. Аналогично, следующие два вызова приводят к выводу Уровень 3: и Уровень 4:.

Когда достигнут уровень 4, переменная n равна 4, поэтому проверка в if не проходит. Функция up_and_down() не вызывается снова. Вместо этого вызов уровня 4 продолжается выполнением второго оператора вывода, который отображает строку УРОВЕНЬ 4:, т.к. переменная n имеет значение 4. В этой точке вызов уровня 4 заканчивается, а управление возвращается функции, которая его инициировала (вызов уровня 3). Последним оператором, выполненным внутри вызова уровня 3, был вызов уровня 4 в операторе if. Следовательно, выполнение уровня 3 возобновляется со следующего оператора, которым является второй оператор вывода. Это приводит к отображению строки УРОВЕНЬ 3:. Затем уровень 3 завершается, передавая управление уровню 2, который выводит строку УРОВЕНЬ 2:, и т.д.

Обратите внимание, что на каждом уровне рекурсии применяется собственная закрытая переменная n. Этот факт легко установить, взглянув на значения адресов. (Конечно, в общем случае в разных системах адреса будут отличаться и возможно иметь другие форматы. Важный момент заключается в том, что адрес в строке Уровень 1: совпадает с адресом в строке УРОВЕНЬ 1: и т.д.)

Если вы находите приведенное объяснение слегка запутанным, то представьте, что имеется цепочка вызовов функций, в которой fun1() вызывает fun2(), fun2() вызывает fun3() и fun3() вызывает fun4(). Когда fun4() завершается, управление передается fun3(). По завершении fun3() управление передается fun2(). Когда заканчивается fun2(), управление возвращается обратно fun1(). Рекурсивный случай работает точно так же, за исключением того, что функции fun1(), fun2(), fun3() и fun4() являются одной и той же функцией.

Основы рекурсии

Поначалу рекурсия может быть непонятной, поэтому давайте рассмотрим несколько базовых аспектов, которые помогут понять процесс.

Во-первых, каждый уровень вызова функции имеет собственные переменные. То есть переменная n уровня 1 отличается от переменной n уровня 2, так что программа создает четыре разных переменных, каждая из которых имеет имя n и собственное значение, отличающееся от других. Когда в конечном итоге программа возвращается к вызову функции up_and_down() первого уровня, исходная переменная n по-прежнему имеет значение 1, с которого она начинала (рис. 9.4).

Во-вторых, каждый вызов функции сбалансирован с возвратом. Когда поток управления достигает оператора return в конце последнего уровня рекурсии, управление переходит на предыдущий уровень рекурсии.

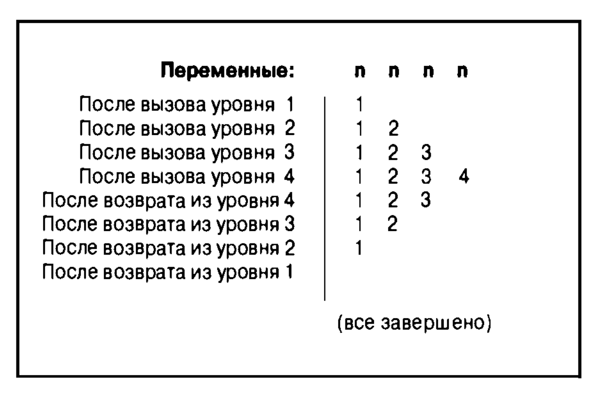


Рис. 9.4. Переменные рекурсии

Переход сразу же к первоначальному вызову внутри main() не происходит. Вместо этого управление должно пройти через каждый уровень рекурсии, возвращаясь с одного уровня up_and_down() на уровень функции up_and_down(), которая ее вызвала.

В-третьих, операторы в рекурсивной функции, которые предшествуют рекурсивному вызову, выполняются в том же самом порядке, в каком эти функции вызывались. Например, в листинге 9.6 первый оператор вывода находится перед рекурсивным вызовом. Он был выполнен четыре раза в порядке следования рекурсивных вызовов: уровень 1, уровень 2, уровень 3 и уровень 4.

В-четвертых, операторы в рекурсивной функции, которые находятся после рекурсивного вызова, выполняются в порядке, обратном тому, в каком эти функции вызывались. Например, второй оператор вывода располагается после рекурсивного вызова, и он выполнялся в следующем порядке: уровень 4, уровень 3, уровень 2, уровень 1. Это свойство рекурсии полезно при программировании задач, предусматривающих изменение порядка на противоположный. Вскоре вы увидите пример.

В-пятых, хотя каждый уровень рекурсии обладает собственным набором переменных, сам код не дублируется. Код – это последовательность инструкций, а вызов функции представляет собой команду перехода на начало этой последовательности инструкций. Рекурсивный вызов затем возвращает программу в начало упомянутой последовательности инструкций. Если не обращать внимания на то, что рекурсивные вызовы создают новые переменные при каждом вызове, они во многом напоминают цикл. На самом деле временами рекурсия может быть использована вместо цикла и наоборот.

Наконец, в-шестых, очень важно, чтобы рекурсивная функция содержала код, который мог бы остановить последовательность рекурсивных вызовов. Обычно в рекурсивной функции применяется проверка if или ее эквивалент для прекращения рекурсии, когда какой-то параметр функции достигает определенного значения. Чтобы это работало, в каждом вызове должно использоваться отличающееся значение для этого параметра. В последнем примере функция up_and_down(n) вызывает up_and_down(n+1). В итоге фактический аргумент достигает значения 4 и проверка условия if (n < 4) не проходит.

Хвостовая рекурсия

В простейшей форме рекурсии рекурсивный вызов находится в конце функции, непосредственно перед оператором return. Такая рекурсия называется *хвостовой* или *концевой*, потому что рекурсивный вызов производится в конце. Хвостовая рекурсия является простейшей формой рекурсии, поскольку она действует подобно циклу.

Давайте рассмотрим версии с циклом и хвостовой рекурсией для функции вычисления факториала. *Факториал* целого числа – это результат произведения всех целых чисел, начиная с 1 и заканчивая заданным числом. Например, факториал 3 (записывается как 3!) соответствует произведению 1*2*3. Кроме того, 0! принимается равным 1, а для отрицательных чисел факториалы не определены. В листинге 9.7 в одной функции для вычисления факториала применяется цикл for, а во второй функции – рекурсия.

Листинг 9.7. Программа factor.c

```
// factor.c -- использует циклы и рекурсию для вычисления факториалов
#include <stdio.h>
long fact(int n);
long rfact(int n);
int main(void)
{
    int num;
    printf("Эта программа вычисляет факториалы.\n");
```

```
    printf("Введите значение в диапазоне 0-12 (q для завершения):\n");
    while (scanf("%d", &num) == 1)
    {
        if (num < 0)
            printf("Отрицательные числа не подходят.\n");
        else if (num > 12)
            printf("Вводимое значение должно быть меньше 13.\n");
        else
        {
            printf("цикл: факториал %d = %ld\n",
                    num, fact(num));
            printf("рекурсия: факториал %d = %ld\n",
                    num, rfact(num));
        }
        printf("Введите значение в диапазоне 0-12 (q для завершения):\n");
    }
    printf("Программа завершена.\n");
    return 0;
}
long fact(int n)     // функция, основанная на цикле
{
    long ans;
    for (ans = 1; n > 1; n--)
        ans *= n;
    return ans;
}
long rfact(int n)    // рекурсивная версия
{
    long ans;
    if (n > 0)
        ans= n * rfact(n-1);
    else
        ans = 1;
    return ans;
}
```

Программа тестового драйвера ограничивает входные данные целыми значениями в диапазоне от 0 до 12. Оказывается, что значение 12! немного меньше полумиллиарда, поэтому результат 13! занимает значительно больше памяти, чем тип long в нашей системе. Для вычисления факториалов, превосходящих 12!, придется использовать тип большего размера, такой как double или long long.

Ниже приведены результаты пробного запуска:

```
Эта программа вычисляет факториалы.
Введите значение в диапазоне 0-12 (q для завершения):
5
цикл: факториал 5 = 120
рекурсия: факториал 5 = 120
Введите значение в диапазоне 0-12 (q для завершения):
10
цикл: факториал 10 = 3628800
рекурсия: факториал 10 = 3628800
Введите значение в диапазоне 0-12 (q для завершения):
q
Программа завершена.
```

Версия с циклом инициализирует переменную ans значением 1, а затем умножает ее на целые числа от n до 2. Формально следовало бы умножить также и на 1, но результат от этого не изменится.

Теперь взглянем на рекурсивную версию. Ключевым моментом является уравнение n! = n × (n-1)!. Оно следует из того факта, что (n-1)! представляет собой произведение всех положительных целых чисел до n-1. Таким образом, умножение его на n дает произведение целых чисел вплоть до n. Это хорошо вписывается в рекурсивный подход. Если вы назовете функцию rfact(), то rfact(n) соответствует n * rfact(n-1). Следовательно, вычислить значение rfact(n) можно, вызвав в ней rfact(n-1), как делается в листинге 9.7. Разумеется, вы должны прервать рекурсию в какой-то точке, и это можно сделать, установив возвращаемое значение в 1, когда n равно 0.

Рекурсивная версия программы в листинге 9.7 дает тот же самый результат, что и версия с циклом. Обратите внимание, что хотя вызов rfact() не является последней строкой в функции, это последний оператор, выполняемый, когда n > 0, т.е. мы имеем дело с хвостовой рекурсией.

Учитывая возможность применения в коде функции либо цикла, либо рекурсии, какому подходу должно отдаваться предпочтение? Обычно цикл является более удачным выбором. Во-первых, из-за того, что каждый рекурсивный вызов создает собственный набор переменных, вариант с рекурсией использует больше памяти; каждый рекурсивный вызов помещает в стек новый набор переменных. При этом ограниченный объем стека может устанавливать предел количества рекурсивных вызовов. Во-вторых, рекурсия выполняется медленнее, т.к. каждый вызов функции занимает определенное время. Для чего тогда демонстрировался этот пример? Причина в том, что хвостовая рекурсия является самой простой формой рекурсии для ее понимания, а рекурсия заслуживает освоения, поскольку в ряде случаев простая альтернатива в виде цикла отсутствует.

Рекурсия и изменение порядка на противоположный

Давайте теперь рассмотрим задачу, для которой способность рекурсии изменять порядок на противоположный оказывается полезной. (Это случай, когда рекурсия проще, чем применение цикла.) Задача заключается в написании функции, которая выводит двоичный эквивалент целого числа. В двоичной записи числа представляются степенями 2. Подобно тому, как 234 в десятичном виде означает $2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$, число 101 в двоичном виде $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$. В двоичных числах используются только цифры 0 и 1.

Для решения задачи необходим некоторый метод, или *алгоритм*. Скажем, каким образом можно найти двоичный эквивалент 5? Ясно, что нечетные числа должны иметь двоичное представление, оканчивающееся цифрой 1. Четные числа оканчиваются цифрой 0, поэтому вы можете определить, является последняя цифра 1 или 0, вычислив значение 5 % 2. Если результат равен 1, то число 5 нечетное, и последней цифрой будет 1. В общем случае, если n — число, то последней цифрой будет n % 2, поэтому первая найденная цифра — это последняя цифра, которую нужно вывести. Это предполагает применение рекурсивной функции, в которой выражение n % 2 вычисляется до рекурсивного вызова, но результат выводится после него. Таким образом, первое вычисленное значение является последним выводимым значением.

Чтобы получить следующую цифру, разделите исходное число на 2. Это двоичный эквивалент сдвига десятичной точки на одну позицию влево, что позволит выяснить следующую двоичную цифру. Если получается четное значение, то следующей двоичной цифрой будет 0, а если нечетное — то 1. Например, 5/2 дает 2 (целочисленное деление), так что следующая цифра — 0. Теперь мы имеем 01. Далее повторим этот

процесс, разделив 2 на 2, чтобы получить 1. Вычисление 1 % 2 дает 1, поэтому следующей цифрой будет 1. В результате имеем 101. Когда вы должны остановиться? Вы останавливаетесь, когда результат деления на 2 оказывается меньше 2, поскольку пока он остается равным 2 или больше, существует еще одна двоичная цифра. Каждое деление на 2 сокращает на одну двоичную цифру, пока не будет достигнут конец. (Если это выглядит запутанным, обратитесь к десятичной аналогии. Остатком от деления 628 на 10 является 8, следовательно, 8 – последняя цифра. Целочисленное деление на 10 дает 62, а остаток от деления 62 на 10 равен 2, поэтому следующей цифрой будет 2 и т.д.) Описанный подход реализован в листинге 9.8.

Листинг 9.8. Программа binary.c

```
/* binary.c -- выводит целые числа в двоичной форме */
#include <stdio.h>
void to_binary(unsigned long n);
int main(void)
{
   unsigned long number;
   printf("Введите целое число (q для завершения):\n");
   while (scanf("%lu", &number) == 1)
   {
      printf("Двоичный эквивалент: ");
      to_binary(number);
      putchar('\n');
      printf("Введите целое число (q для завершения):\n");
   }
   printf("Программа завершена.\n");
   return 0;
}
void to_binary(unsigned long n)  /* рекурсивная функция */
{
   int r;
   r = n % 2;
   if (n >= 2)
      to_binary(n / 2);
   putchar(r == 0 ? '0' : '1');
   return;
}
```

Функция to_binary() должна отображать символ '0', если числовое значение переменной r равно 0, и '1', если оно равно 1. Условное выражение r == 0 ? '0' : '1' обеспечивает такое преобразование числовых значений в символьные.

Ниже показан результаты пробного запуска:

```
Введите целое число (q для завершения):
9
Двоичный эквивалент: 1001
Введите целое число (q для завершения):
255
Двоичный эквивалент: 11111111
Введите целое число (q для завершения):
1024
Двоичный эквивалент: 10000000000
Введите целое число (q для завершения):
q
Программа завершена.
```

Можно ли воспользоваться этим алгоритмом для вычисления двоичного представления числа без применения рекурсии? Да, это возможно. Но поскольку данный алгоритм первой вычисляет последнюю цифру, перед отображением результата все цифры пришлось бы где-то сохранять (например, в массиве). В главе 15 будет представлен пример нерекурсивного подхода.

Преимущества и недостатки рекурсии

Рекурсия обладает как преимуществами, так и недостатками. Одно из преимуществ рекурсии состоит в том, что она предлагает простейшее решение ряда задач программирования. Один из недостатков заключается в том, что некоторые рекурсивные алгоритмы могут быстро исчерпать ресурсы памяти компьютера. Кроме того, рекурсию трудно документировать и сопровождать. Рассмотрим пример, который иллюстрирует и преимущества, и недостатки рекурсии.

Числа Фибоначчи можно определить следующим образом: первое число Фибоначчи – это 1, второе число Фибоначчи – тоже 1, а каждое последующее число Фибоначчи представляет собой сумму двух предшествующих чисел. Таким образом, первые несколько чисел в последовательности выглядят так: 1, 1, 2, 3, 5, 8, 13. Числа Фибоначчи пользуются особой любовью у математиков; издается даже специальный журнал, посвященный таким числам. Однако не будем углубляться в это. Давайте создадим функцию, которая для заданного положительного целого числа n возвращает соответствующее число Фибоначчи.

Вначале отметим преимущество рекурсии: она обеспечивает простое определение. Если мы назовем функцию `Fibonacci()`, то `Fibonacci(n)` должна возвращать значение 1, если n равно 1 или 2, и сумму `Fibonacci(n-1)` и `Fibonacci(n-2)` в противном случае:

```
unsigned long Fibonacci(unsigned n)
{
   if (n > 2)
      return Fibonacci(n-1) + Fibonacci(n-2);
   else
      return 1;
}
```

Рекурсивная функция С просто повторяет математическое определение рекурсии. В этой функции используется *двойная рекурсия*, т.е. вызывает себя дважды. Это обстоятельство является источником ее слабости.

Чтобы увидеть природу этой слабости, предположим, что имеется вызов `Fibonacci(40)`. Это будет первый уровень рекурсии, и он выделяет память для переменной по имени n. Затем он вызывает функцию `Fibonacci()` два раза, создавая на втором уровне рекурсии еще две переменных n. Каждый из этих двух вызовов генерирует еще два вызова, которые, в свою очередь, требуют еще четырех переменных с именами n на третьем уровне рекурсии, что в сумме дает семь переменных. На каждом уровне количество переменных удваивается по сравнению с предыдущим уровнем, т.е. объем переменных возрастает по экспоненте! Как вы могли убедиться на примере с пшеничными зернами в главе 5, экспоненциальное возрастание быстро приводит к огромным значениям. В рассматриваемом случае экспоненциальное возрастание быстро приводит к тому, что компьютеру потребуется гигантский объем памяти, что, скорее всего, приведет к аварийному завершению программы.

На самом деле это экстремальный пример, однако он хорошо иллюстрирует необходимость в соблюдении осторожности во время применения рекурсии, особенно когда важным фактором является эффективность.

Все функции С созданы равными друг другу

Каждая функция С в программе имеет равное положение с остальными функциями. Каждая из них может вызывать любую другую функцию или быть вызванной в других функциях. Это делает функции С несколько отличающимися от процедур в языках Pascal и Modula-2, поскольку упомянутые процедуры могут быть вложенными друг в друга. Процедурам в одном вложении ничего не известно о процедурах в другом вложении.

Не является ли функция `main()` особенной? Да, ее особенность в том, что когда программа, состоящая из нескольких функций, собирается вместе, выполнение начинается с первого оператора в `main()`, но этим ее отличие и ограничивается. Даже функция `main()` может вызывать саму себя рекурсивно или быть вызванной из других функций, хотя подобное встречается редко.

Компиляция программ, состоящих из двух и более файлов исходного кода

Простейший подход к использованию нескольких функций предусматривает их помещение в один файл. Затем нужно просто скомпилировать этот файл, как если бы он содержал единственную функцию. Другие подходы в большей степени зависят от системы, как показано в последующих разделах.

Unix

В этом случае предполагается, что в системе Unix имеется компилятор Unix С — `cc`. (Первоначальный компилятор `cc` был выведен из употребления, но во многих системах Unix команда `cc` служит псевдонимом команды запуска какого-то другого компилятора, обычно `gcc` или `clang`.) Пусть `file1.c` и `file2.c` — два файла, содержащие функции С. Тогда следующая команда скомпилирует оба файла и создаст исполняемый файл по имени `a.out`:

```
cc file1.c file2.c
```

Кроме того, создаются два объектных файла — `file1.o` и `file2.o`. Если позже вы измените содержимое `file1.c`, но не `file2.c`, можно скомпилировать первый файл и объединить его с объектным кодом из второго файла с помощью такой команды:

```
cc file1.c file2.o
```

В Unix имеется команда `make`, которая автоматизирует управление многофайловыми программами, но эта тема выходит за рамки настоящей книги. Обратите внимание, что утилита Terminal в системе OS X открывает среду командной строки Unix, но компиляторы командной строки (GCC и Clang) придется загрузить из веб-сайта Apple.

Linux

В этом случае предполагается, что в системе Linux установлен компилятор GNU С — GCC. Пусть `file1.c` и `file2.c` — два файла, содержащие функции С. Тогда следующая команда скомпилирует оба файла и создаст исполняемый файл по имени `a.out`:

```
gcc file1.c file2.c
```

В дополнение создаются два объектных файла `file1.o` и `file2.o`. Если позже вы измените содержимое файла `file1.c`, но не `file2.c`, можно скомпилировать первый файл и объединить его с объектным кодом из второго файла, посредством такой команды:

```
gcc file1.c file2.o
```

Компиляторы командной строки DOS

Большинство компиляторов командной строки DOS работают аналогично команде cc в Unix, но применяют другое имя. Отличие в том, что объектные файлы получают расширение .obj, а не .o. Некоторые компиляторы вместо файлов с объектным кодом генерируют промежуточные файлы с кодом на языке ассемблера или другим специальным кодом.

Компиляторы интегрированных сред разработки в Windows и Apple

Компиляторы интегрированных сред разработки (integrated development environment – IDE) для операционных систем Windows и Macintosh являются *ориентированными на проекты. Проект* описывает ресурсы, используемые конкретной программой. Эти ресурсы включают и файлы исходного кода. Если вы применяли один из этих компиляторов, то вероятно должны были создавать проекты для выполнения однофайловых программ. В случае многофайловых программ найдите команду меню, которая позволяет добавлять в проект файл исходного кода. Вы должны обеспечить присутствие в проекте всех файлов исходного кода (с расширением .c). Во многих IDE-средах наличие в списке проекта заголовочных файлов (с расширением .h) не обязательно. Идея в том, что проект управляет использованием файлов исходного кода, а с помощью директив #include в этих файлах указываются применяемые заголовочные файлы. Тем не менее, в случае XCode заголовочные файлы должны быть добавлены в проект.

Использование заголовочных файлов

Если поместить функцию main() в один файл, а определения собственных функций – в другой, то в первом файле по-прежнему будут нужны прототипы функций. Вместо того чтобы набирать их каждый раз, когда применяется файл с функциями, прототипы функций можно сохранить в заголовочном файле. Именно это сделано в стандартной библиотеке C путем помещения, к примеру, прототипов функций ввода-вывода в файл stdio.h, а прототипов математических функций – в файл math.h. Со своими файлами функций вы можете поступить аналогично.

Вы также часто будете пользоваться препроцессором C для определения констант, применяемых в программе. Такие определения возможны только для файла, содержащего директивы #define. Если разнести функции, применяемые в программе, по отдельным файлам, то также придется обеспечить доступность каждому файлу директив #define. Наиболее прямолинейный способ предусматривает повторный набор директив в каждом файле, но это требует времени и увеличивает вероятность допущения ошибки при наборе. Вдобавок возникает проблема сопровождения: если вы решите изменить значение в директиве #define, то нужно будет не забыть сделать это в каждом файле. Более разумное решение предполагает размещение директив #define в заголовочном файле с последующим использованием директивы #include в каждом файле исходного кода.

Таким образом, хорошим тоном в программировании считается размещение прототипов функций и определенных констант в заголовочном файле. Давайте рассмотрим пример. Предположим, что вы управляете сетью из четырех отелей. В каждом отеле действует разная плата за номер, но все номера в одном и том же отеле стоят одинаково. Для тех, кто забронировал номер на несколько суток, плата за вторые сутки составляет 95% от платы за первые сутки, третьи сутки оплачиваются в размере

95% от платы за вторые сутки и т.д. Вам нужна программа, которая позволила бы выбрать отель, указать количество суток и рассчитать суммарную стоимость проживания. Желательно, чтобы программа имела меню, которое позволило бы вводить данные до тех пор, пока не будет принято решение выйти из программы.

В листингах 9.9, 9.10 и 9.11 показано, как можно решить эту задачу. Листинг 9.9 содержит функцию `main()`, которая обеспечивает общую организацию для программы. В листинге 9.10 приведен код функций поддержки, хранящийся в отдельном файле. Наконец, в листинге 9.11 представлен заголовочный файл, в котором содержатся определения констант и прототипы функций для всех исходных файлов программы. Вспомните, что в средах Unix и DOS двойные кавычки в директиве `#include "hotels.h"` указывают на то, что включаемый файл хранится в текущем рабочем каталоге (обычно в каталоге, где находится исходный код). В IDE-среде необходимо знать, как заголовочные файлы включаются в проект.

Листинг 9.9. Управляющий модуль usehotel.c

```
/* usehotel.c -- программа определения стоимости номеров */
/* компилируется вместе с листингом 9.10           */
#include <stdio.h>
#include "hotel.h" /* определяет константы, объявляет функции */
int main(void)
{
   int nights;
   double hotel_rate;
   int code;
   while ((code = menu()) != QUIT)
   {
      switch(code)
      {
      case 1 : hotel_rate = HOTEL1;
             break;
      case 2 : hotel_rate = HOTEL2;
             break;
      case 3 : hotel_rate = HOTEL3;
             break;
      case 4 : hotel_rate = HOTEL4;
             break;
      default: hotel_rate = 0.0;
             printf("Ошибка!\n");
             break;
      }
      nights = getnights();
      showprice(hotel_rate, nights);
   }
   printf("Благодарим за использование и желаем успехов.\n");
   return 0;
}
```

Листинг 9.10. Модуль функций поддержки hotel.c

```
/* hotel.c -- функции управления отелем */
#include <stdio.h>
#include "hotel.h"
```

```
int menu(void)
{
   int code, status;
   printf("\n%s%s\n", STARS, STARS);
   printf("Введите число, соответствующее выбранному отелю:\n");
   printf("1) Fairfield Arms          2) Hotel Olympic\n");
   printf("3) Chertworthy Plaza       4) The Stockton\n");
   printf("5) выход\n");
   printf("%s%s\n", STARS, STARS);
   while ((status = scanf("%d", &code)) != 1 ||
   (code < 1 || code > 5))
   {
      if (status != 1)
         scanf("%*s");    // отбрасывание нецелочисленного ввода
      printf("Введите целое число от 1 до 5.\n");
   }
   return code;
}
int getnights(void)
{
   int nights;

   printf("На сколько суток вы бронируете номер? ");
   while (scanf("%d", &nights) != 1)
   {
      scanf("%*s");       // исключение нецелочисленного ввода
      printf("Введите целое число, такое как 2.\n");
   }
   return nights;
}
void showprice(double rate, int nights)
{
   int n;
   double total = 0.0;
   double factor = 1.0;

   for (n = 1; n <= nights; n++, factor *= DISCOUNT)
      total += rate * factor;
   printf("Общая стоимость составляет $%0.2f.\n", total);
}
```

Листинг 9.11. Заголовочный файл hotel.h

```
/* hotel.h -- константы и объявления для программы hotel.c */
#define QUIT 5
#define HOTEL1 180.00
#define HOTEL2 225.00
#define HOTEL3 255.00
#define HOTEL4 355.00
#define DISCOUNT 0.95
#define STARS "**********************************"

// отображает список возможных вариантов
int menu(void);

// возвращает количество суток, на которое бронируется номер
int getnights(void);

// вычисляет стоимость в зависимости от расценок и количества
// забронированных суток и отображает результат
void showprice(double rate, int nights);
```

Ниже показаны результаты пробного запуска:

```
******************************************************************
Введите число, соответствующее выбранному отелю:
1) Fairfield Arms          2) Hotel Olympic
3) Chertworthy Plaza       4) The Stockton
5) выход
******************************************************************
3
На сколько суток вы бронируете номер? 1
Общая стоимость составляет $255.00.
******************************************************************
Введите число, соответствующее выбранному отелю:
1) Fairfield Arms          2) Hotel Olympic
3) Chertworthy Plaza       4) The Stockton
5) выход
******************************************************************
4
На сколько суток вы бронируете номер? 3
Общая стоимость составляет $1012.64.
******************************************************************
Введите число, соответствующее выбранному отелю:
1) Fairfield Arms          2) Hotel Olympic
3) Chertworthy Plaza       4) The Stockton
5) выход
******************************************************************
5
Благодарим за использование и желаем успехов.
```

Кстати, эта программа сама по себе обладает рядом интересных особенностей. В частности, функции `menu()` и `getnights()` пропускают нечисловые данные, проверяя возвращаемое значение функции `scanf()` и применяя вызов `scanf("%*s")` для пропуска следующего пробельного символа. Взгляните, как следующий фрагмент функции `menu()` производит проверку на предмет нечислового ввода и ввода числовых значений, выходящих за пределы установленного диапазона:

```
while ((status = scanf("%d", &code)) != 1  ||
       (code < 1 || code > 5))
```

В этом фрагменте кода используется действующее в C правило о том, что логические выражения вычисляются слева направо и что вычисление прекращается, как только становится понятно, что выражение ложно. В данном случае значения `code` проверяются только после того, как будет установлено, что функция `scanf()` успешно прочитала целочисленное значение.

Назначение отдельных задач разным функциям способствует такому улучшению кода. На первом этапе написания программы функция `menu()` или `getnights()` может применять функцию `scanf()` без дополнительных функций проверки допустимости данных. Затем, когда базовая версия заработает, можно приступить к совершенствованию отдельных модулей.

Выяснение адресов: операция &

Одним из наиболее важных концепций языка C (и временами самой трудной для понимания) является *указатель*, который представляет собой переменную, используемую для хранения адреса. Вы уже видели, что функция `scanf()` работает с адресами

аргументов. В общем случае любая функция С, которая изменяет значение в вызывающей функции без использования значения в `return`, применяет адреса. Далее мы рассмотрим функции, использующие адреса, и начнем с унарной операции `&`. (В следующей главе мы продолжим исследование и работу с указателями.)

Унарная операция `&` предоставляет адрес, по которому хранится переменная. Если `pooh` является именем переменной, то `&pooh` – адрес этой переменной. Об адресе можно думать как о ячейке в памяти. Пусть имеется следующий оператор:

```
pooh = 24;
```

Предположим, что адресом, по которому хранится переменная `pooh`, является `0B76`. (В IBM PC адреса часто задаются в виде шестнадцатеричных значений.) Тогда оператор

```
printf("%d %p\n", pooh, &pooh);
```

выведет следующий результат (`%p` – спецификатор для адресов):

```
24 0B76
```

В листинге 9.12 эта операция применяется, чтобы посмотреть, где хранятся переменные, которые имеют одно и то же имя, но используются в разных функциях.

Листинг 9.12. Программа `loccheck.c`

```
/* loccheck.c -- проверка для выяснения, где хранятся переменные */
#include <stdio.h>
void mikado(int);                    /* объявление функции */
int main(void)
{
    int pooh = 2, bah = 5;           /* локальные для main() */

    printf("Внутри main() pooh = %d и &pooh = %p\n",
           pooh, &pooh);
    printf("Внутри main() bah = %d и &bah = %p\n",
           bah, &bah);
    mikado(pooh);

    return 0;
}

void mikado(int bah)                 /* определение функции */
{
    int pooh = 10;                   /* локальная для mikado() */
    printf("Внутри mikado() pooh = %d и &pooh = %p\n",
           pooh, &pooh);
    printf("Внутри mikado() bah = %d и &bah = %p\n",
           bah, &bah);
}
```

Для вывода адресов в листинге 9.12 указан формат `%p` из ANSI С. В нашей системе был получен следующий вывод:

```
Внутри main() pooh = 2 и &pooh = 0x7fff5fbff8e8
Внутри main() bah = 5 и &bah = 0x7fff5fbff8e4
Внутри mikado() pooh = 10 и &pooh = 0x7fff5fbff8b8
Внутри mikado() bah = 2 и &bah = 0x7fff5fbff8bc
```

Способ представления адреса спецификатором `%p` варьируется между реализациями. Однако многие реализации вроде той, что применяется в этом примере, отображают адрес в шестнадцатеричной форме. Учитывая, что каждая шестнадцатеричная

цифра соответствует четырем битам, показанные адреса, состоящие из 12 цифр, соответствуют 40-битным адресам.

Что демонстрирует приведенный вывод? Во-первых, две переменные pooh имеют отличающиеся адреса. То же самое справедливо и для двух переменных bath. Таким образом, как было обещано ранее, компьютер рассматривает их как четыре разных переменных. Во-вторых, вызов mikado(pooh) передает значение (2) фактического аргумента (pooh из main()) формальному аргументу (bah из mikado()). Обратите внимание, что было передано просто значение. Обе переменные (pooh из main() и bah из mikado()) сохраняют свою идентичность.

Второй аспект был отмечен особо из-за того, что так дело обстоит не во всех языках. Например, в FORTRAN подпрограмма воздействует на исходную переменную в вызывающей процедуре. Переменная подпрограммы может иметь другое имя, но адрес будет тем же. В языке С это не так. Каждая функция использует собственные переменные. Такой подход предпочтительнее, поскольку он предотвращает загадочное изменение исходной переменной вследствие какого-то побочного эффекта вызванной функции. Тем не менее, этот подход также может создавать определенные трудности, как будет показано в следующем разделе.

Изменение переменных в вызывающей функции

Временами необходимо внести изменения в переменные другой функции. Например, распространенная задача, связанная с сортировкой, предусматривает обмен значениями двух переменных. Предположим, что есть две переменных с именами x и y, и нужно, чтобы они обменялись значениями. Простая последовательность

```
x = y;
y = x;
```

проблему не решит, потому что к моменту достижения второй строки исходное значение x уже было заменено исходным значением y. Необходим дополнительный оператор для временного запоминания исходного значения x:

```
temp = x;
x = y;
y = temp;
```

Теперь, когда метод заработал, вы можете поместить его в функцию и создать драйвер для ее тестирования. Чтобы прояснить, какая переменная принадлежит main(), а какая — interchange(), в листинге 9.13 функция main() имеет дело с переменными x и y, а interchange() — с переменными u и v.

Листинг 9.13. Программа swap1.c

```
/* swap1.c -- первая попытка создания функции обмена значениями */
#include <stdio.h>
void interchange(int u, int v); /* объявление функции */
int main(void)
{
   int x = 5, y = 10;
   printf("Первоначально x = %d и y = %d.\n", x , y);
   interchange(x, y);
   printf("Теперь x = %d и y = %d.\n", x, y);
   return 0;
}
```

```
void interchange(int u, int v) /* определение функции */
{
   int temp;

   temp = u;
   u = v;
   v = temp;
}
```

Выполнение этой программы дает следующие результаты:

```
Первоначально x = 5 и y = 10.
Теперь x = 5 и y = 10.
```

Как видите, значения не поменялись. Чтобы посмотреть, что именно пошло не так, давайте добавим в функцию `interchange()` несколько операторов вывода (листинг 9.14).

Листинг 9.14. Программа swap2.c

```
/* swap2.c -- исследование программы swap1.c */
#include <stdio.h>
void interchange(int u, int v);

int main(void)
{
   int x = 5, y = 10;

   printf("Первоначально x = %d и y = %d.\n", x , y);
   interchange(x, y);
   printf("Теперь x = %d и y = %d.\n", x, y);

   return 0;
}

void interchange(int u, int v)
{
   int temp;

   printf("Первоначально u = %d и v = %d.\n", u , v);
   temp = u;
   u = v;
   v = temp;
   printf("Теперь u = %d и v = %d.\n", u , v);
}
```

Вот новый вывод:

```
Первоначально x = 5 и y = 10.
Первоначально u = 5 и v = 10.
Теперь u = 10 и v = 5.
Теперь x = 5 и y = 10.
```

С функцией `interchange()` все в порядке; она меняет местами значения `u` и `v`. Проблема возникает при передаче результатов в `main()`. Как было указано, в `interchange()` применяются переменные, отличающиеся от используемых в `main()`, так что обмен значениями между `u` и `v` не влияет на переменные `x` и `y`. Может быть, каким-то образом применить `return`?

Что же, функцию `interchange()` можно было бы завершить строкой

```
return(u);
```

а затем заменить ее вызов в main() следующим образом:

```
x = interchange(x,y);
```

Это изменение приводит к тому, что x получает новое значение, но значение y остается незатронутым. С помощью return обратно в вызывающую функцию можно отправлять только одно значение, а нам нужно передать два значения. Но все-таки это можно сделать! Понадобится только воспользоваться указателями.

Указатели: первое знакомство

Что собой представляют указатели? По существу *указатель* — это переменная (или в общем случае объект данных), значением которой является адрес в памяти. Подобно тому, как переменная char имеет в качестве значения символ, а переменная int — целое число, переменная типа указателя содержит значение адреса. С указателями связаны многочисленные применения в языке C; в данной главе вы увидите, как и почему они применяются как параметры функций.

Если назначить переменной типа указателя имя ptr, можно записывать операторы вроде показанного ниже:

```
ptr = &pooh;  // присваивает переменной ptr адрес переменной pooh
```

Мы говорим, что ptr "указывает на" pooh. Разница между ptr и &pooh состоит в том, что ptr является переменной, а &pooh — константой. Иначе говоря, ptr — это модифицируемое l-значение, а &pooh — r-значение. При желании можно сделать так, чтобы переменная ptr указывала на что-то другое:

```
ptr = &bah;   // переменная ptr указывает на bah вместо pooh
```

Теперь значением ptr будет адрес bah.

Чтобы создать переменную-указатель, вы должны иметь возможность объявить ее тип. Предположим, что вы хотите объявить переменную ptr так, чтобы она могла хранить адрес значения int. Для такого объявления необходимо использовать новую операцию, которую мы рассмотрим в следующем разделе.

Операция разыменования: *

Предположим, вам известно, что ptr указывает на bah:

```
ptr = &bah;
```

Тогда для выяснения значения, хранящегося в переменной bah, можно применить операцию *разыменования* * (которая также называется операцией *снятия косвенности*); не путайте эту унарную операцию с бинарной операцией умножения * (тот же символ, но другой синтаксис):

```
val = *ptr;   // выяснение значения, на которое указывает ptr
```

Операторы ptr = &bah; и val = *ptr; вместе эквивалентны следующему оператору:

```
val = bah;
```

Использование операций взятия адреса и снятия косвенности — это косвенный путь достижения нужного результата, откуда и происходит название "операция снятия косвенности".

Сводка: операции, связанные с указателями

Операция взятия адреса

&

Общий комментарий

Символ &, за которым следует имя переменной, предоставляет адрес этой переменной.

Пример

&nurse является адресом переменной nurse.

Операция разыменования

*

Общий комментарий

Символ *, за которым следует имя указателя или адрес, предоставляет значение, которое хранится по указанному адресу.

Пример

```
nurse = 22;
ptr = &nurse;           // указатель на nurse
val = *ptr;             // присваивает val значение, хранящееся в ячейке ptr
```

В конечном итоге переменная val получает значение 22.

Объявление указателей

Вы уже знаете, как объявлять переменные int и других фундаментальных типов. А как объявить переменную типа указателя? Можно было бы предположить, что объявление выглядит примерно так:

```
pointer ptr;            // указатель не объявляется подобным образом
```

Почему это не подойдет? Дело в том, что объявления переменной указателем далеко не достаточно. Должен быть также задан вид переменной, на которую указывает указатель. Причина в том, что разные типы переменных занимают разные объемы памяти, а некоторые операции с указателями требуют знания этих размеров памяти. Кроме того, программе должно быть известно, данные какого вида хранятся по конкретному адресу. Типы long и float могут занимать один и тот же объем памяти, но хранят числа они по-разному. Ниже демонстрируются случаи объявления указателей:

```
int * pi;               // pi - указатель на целочисленную переменную
char * pc;              // pc - указатель на символьную переменную
float * pf, * pg;       // pf, pg - указатели на переменные с плавающей запятой
```

Спецификация типа идентифицирует тип переменной, на которую указывает указатель, а звездочка (*) — что переменная сама является указателем. Объявление int * pi; говорит о том, что pi является указателем, и *pi имеет тип int (рис. 9.5).

Пробел между символом * и именем указателя необязателен. Часто программисты применяют пробел в объявлении и опускают его при разыменовании переменной.

Значение (*pc), на которое указывает указатель pc, имеет тип char. А что собой представляет собственно pc? Мы описываем его как имеющий тип "указатель на char". Значение pc — это адрес, и в большинстве систем он внутренне представлен как целое число без знака. Однако вы не должны считать, что указатель относится к целочисленному типу. Есть действия, которые можно выполнять над целыми числами, но нельзя — над указателями, и наоборот.

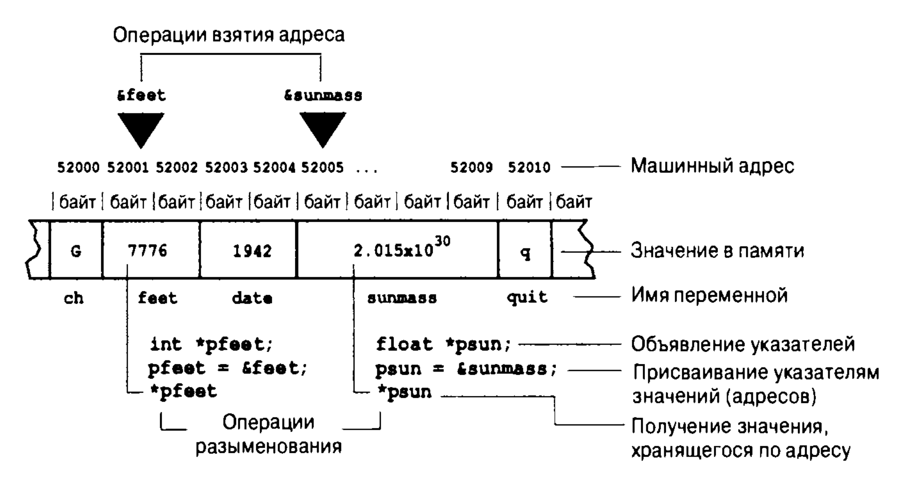


Рис. 9.5. Объявление и использование указателей

Например, целые числа можно умножать, но делать это в отношении указателей не допускается. Таким образом, указатель в действительности представляет собой новый тип, а не целочисленный. По этой причине, как упоминалось ранее, в ANSI C специально для указателей предусмотрена форма %p.

Использование указателей для обмена данными между функциями

Мы лишь слегка коснулись многообразного и удивительного мира указателей, но нас интересует применение указателей для решения задачи обмена данными. В листинге 9.15 представлена программа, в которой с помощью указателей обеспечивается работа функции `interchange()`. Давайте рассмотрим код, запустим ее и выясним, как она работает.

Листинг 9.15. Программа swap3.c

```
/* swap3.c -- использование указателей для обмена значениями переменных */
#include <stdio.h>
void interchange(int * u, int * v);

int main(void)
{
   int x = 5, y = 10;

   printf("Первоначально x = %d и y = %d.\n", x, y);
   interchange(&x, &y);     // передача адресов в функцию
   printf("Теперь x = %d и y = %d.\n", x, y);

   return 0;
}

void interchange(int * u, int * v)
{
   int temp;

   temp = *u;               // temp получает значение, на которое указывает u
   *u = *v;
   *v = temp;
}
```

Заработает ли программа из листинга 9.15 после компиляции?

```
Первоначально x = 5 и y = 10.
Теперь x = 10 и y = 5.
```

Да, она работает.

Давайте теперь проанализируем код из листинга 9.15. Прежде всего, вызов функции выглядит так:

```
interchange(&x, &y);
```

Вместо передачи *значений* x и y функция передаются их *адреса*. Это означает, что формальные аргументы u и v, указанные в прототипе и определении функции interchange(), в качестве своих значений будут содержать адреса. Следовательно, они должны быть объявлены как указатели. Поскольку x и y являются целочисленными, а u и v — указателями на целочисленные значения, они объявляются следующим образом:

```
void interchange (int * u, int * v)
```

Далее в теле функции содержится объявление, которое предоставляет область памяти, необходимую для временного хранения:

```
int temp;
```

Для сохранения значения x в temp используется оператор

```
temp = *u;
```

Вспомните, что u имеет значение &x, поэтому u указывает на x. Это означает, что *u дает значение x, что и требовалось. Не следует записывать

```
temp = u;  /* Неправильно */
```

поскольку в этом случае переменной temp присваивается адрес переменной x вместо ее значения, а задача состоит в том, чтобы осуществить обмен значениями, но не адресами.

Аналогично, чтобы присвоить переменной x значение переменной y, применяйте оператор

```
*u = *v;
```

который, в конечном счете, дает следующий результат:

```
x = y;
```

Итак, подведем итоги рассмотренного примера. Нам была нужна функция, которая меняет между собой значения переменных x и y. Передавая в функцию адреса x и y, мы предоставляем interchange() доступ к этим переменным. Используя указатели и операцию *, эта функция может выяснить значения, хранящиеся в этих ячейках, и изменить их.

В прототипе ANSI C имена переменных можно не указывать. Тогда объявление в прототипе будет выглядеть так:

```
void interchange(int *, int *);
```

В общем случае в функцию можно передать два вида информации о переменной. Если вызов имеет следующий вид, то в функцию передается значение переменной x:

```
function1(x);
```

Если же вызвать функцию, как показано ниже, в нее передается адрес переменной x:

```
function2(&x);
```

Первая форма требует, чтобы определение функции включало формальный аргумент того же типа, что и x:

```
int function1(int num)
```

Во второй форме определение функции должно включать формальный параметр, который является указателем на корректный тип:

```
int function2(int * ptr)
```

Применяйте первую форму, если функции необходимо передать значение для какого-то вычисления или действия. Используйте вторую форму, если функция должна изменять значения переменных из вызывающей функции. Все это вы уже делали с функцией `scanf()`. Когда необходимо прочитать значение для переменной (например, `num`), вы применяете вызов `scanf("%d", &num)`. Функция читает значение и затем использует адрес для сохранения значения.

Указатели дают возможность обойти тот факт, что переменные внутри `interchange()` являются локальными. Они позволяют этой функции изменять то, что хранится в `main()`.

Пользователи, знакомые с языками Pascal и Modula-2, могут заметить, что первая форма аналогична параметру-значению, а вторая форма подобна (но не идентична) параметру-переменной в Pascal. Пользователи C++ узнают переменные указатели и заинтересуются, не имеет ли C подобно языку C++ также и ссылочные переменные. Ответ на этот вопрос отрицателен. Пользователям, работающим с BASIC, все это может показаться несколько нарушающим общий порядок. Если материал данного раздела показался непонятным, будьте уверены, что после небольшой практики применение указателей станет простым, обычным и удобным делом (рис. 9.6).

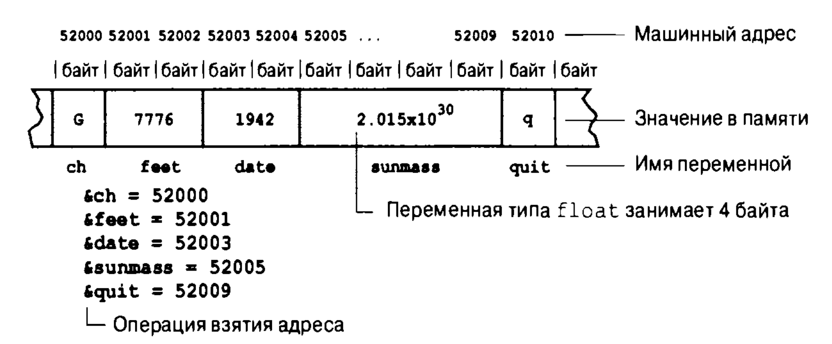


Рис. 9.6. Имена, адреса и значения в системе байтовой адресации, такой как IBM PC

Переменные: имена, адреса и значения

Предыдущее обсуждение указателей было тесно связано с взаимоотношениями между именами, адресами и значениями переменных. Давайте продолжим обсуждение этих аспектов.

При написании программы можно полагать, что переменная имеет два атрибута: имя и значение. (Существуют также и другие атрибуты, в числе которых тип, но это уже другая тема.) После того, как программа скомпилирована и загружена, компьютер также считает, что та же переменная располагает двумя атрибутами: адресом и значением. Адрес является компьютерной версией имени.

Во многих языках адресами занимается компьютер скрыто от программиста. Однако в языке C к адресу можно получить доступ посредством операции `&`. Например, `&barn` — это адрес переменной `barn`.

Значение можно получить из имени, для чего нужно просто указать имя. Например, `printf("%d\n", barn)` выводит значение переменной `barn`.

Значение переменной можно также получить из адреса, воспользовавшись операцией `*`. Например, в операторе `pbarn = &barn;` конструкция `*pbarn` — это значение, хранящееся по адресу `&barn`.

Короче говоря, обычная переменная делает значение главной величиной, а адрес — производной величиной, доступной через операцию `&`. Переменная типа указателя делает адрес главной величиной, а значение — производной величиной, доступной посредством операции `*`.

Хотя ради любопытства вы можете выводить адреса, основное назначение операция `&` состоит не в этом. Гораздо важнее то, что с помощью операций `&`, `*` и указателей становится возможным символическое манипулирование адресами и их содержимым, как было продемонстрировано в программе `swap3.c` (листинг 9.15).

Сводка: функции

Форма

Типичное определение функции в стандарте ANSI C имеет следующую форму:

возвращаемый_тип имя (список объявлений параметров)
тело функции

Список объявлений параметров — это список объявлений переменных, разделенных запятыми. Переменные, отличные от параметров функции, объявляются внутри тела, ограниченного фигурными скобками.

Пример

```
int diff(int x, int y)      // стандарт ANSI C
{                           // начало тела функции
   int z;                   // объявление локальной переменной
   z = x - y;
   return z;                // возвращение значения
}                           // конец тела функции
```

Обмен значениями

Аргументы используются для передачи значений из вызывающей функции в вызванную. Если переменные `a` и `b` имеют значения 5 и 2, то вызов

```
c = diff(a,b);
```

передает переменным `x` и `y` величины 5 и 2. Значения 5 и 2 называются *фактическими аргументами*, а переменные `x` и `y` в функции `diff()` — *формальными параметрами*. Ключевое слово `return` передает одно значение из функции в вызывающую функцию. В приведенном примере `c` получает значение переменной `z`, которое равно 3. Обычно функция не воздействует на переменные в вызывающей функции. Чтобы напрямую влиять на переменные в вызывающей функции, применяйте указатели в качестве аргументов. Это может понадобиться, если в вызывающую функцию нужно возвратить больше одного значения.

Возвращаемый тип функции

Возвращаемый тип функции указывает тип значения, возвращаемого функцией. Если возвращаемое значение имеет тип, отличающийся от объявленного возвращаемого типа, значение приводится к объявленному типу.

Сигнатура функции

Возвращаемый тип функции вместе со списком параметров функции образуют сигнатуру функции. Таким образом, сигнатура указывает типы значений, которые поступают в функцию, и тип значения, передаваемого из нее.

Пример

```
double duff(double, int);      // прототип функции
int main(void)
{
   double q, x;
   int n;
   ...
   q = duff(x,n);              // вызов функции
   ...
}
double duff(double u, int k)   // определение функции
{
   double tor;
   ...
   return tor;                 // возврат значения типа double
}
```

Ключевые понятия

Чтобы успешно и эффективно программировать на С, необходимо понимать, как работают функции. Полезно и даже важно организовывать крупные программы в виде совокупности функций. Если придерживаться практики решения в одной функции только одной задачи, программу легче будет понять и отладить. Разберитесь в том, как функции обмениваются информацией друг с другом, т.е. убедитесь, что вы понимаете, как работают аргументы и возвращаемые значения функций. Кроме того, помните о том, что параметры и другие локальные переменные являются закрытыми для функции. Таким образом, объявление двух переменных с одним и тем же именем в разных функциях приводит к созданию двух разных переменных. Вдобавок одна функция не имеет прямого доступа к переменным, объявленным в другой функции. Такой ограниченный доступ помогает обеспечить целостность данных. Тем не менее, если нужно, чтобы одна функция имела доступ к данным в другой функции, можно использовать аргументы типа указателей.

Резюме

Применяйте функции в качестве строительных блоков для крупных программ. Каждая функция должна иметь единственное четко определенное назначение. Используйте аргументы для передачи значений в функцию и ключевое слово `return` для передачи значения обратно. Если функция возвращает значение не типа `int`, то тип функции должен быть указан в ее определении и в разделе объявлений вызывающей функции. Если необходимо, чтобы функция воздействовала на переменные в вызывающей функции, применяйте адреса и указатели.

Стандарт ANSI С предлагает механизм *прототипирования функций* – мощное расширение языка, которое позволяет компиляторам проверять, корректно ли количество аргументов и правильно ли указаны их типы при вызове функции.

Функция С может вызывать саму себя; это называется *рекурсией*. Некоторые задачи программирования сами приспособлены под рекурсивные решения, но рекурсия может быть неэффективной в плане использования памяти и времени выполнения.

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля приведены в приложении А.

1. Чем отличается фактический аргумент от формального параметра?
2. Напишите заголовки ANSI C для указанных ниже функций. Речь идет только о заголовках, но не о телах.
   а. Функция donut() принимает аргумент int и выводит количество нулей, соответствующее значению этого аргумента.
   б. Функция gear() принимает два аргумента int и возвращает значение типа int.
   в. Функция guess() не принимает аргументов и возвращает значение типа int.
   г. Функция stuff_it() принимает значение double и адрес переменной double и запоминает первое значение в заданной ячейке.
3. Напишите заголовки ANSI C для указанных ниже функций. Речь идет только о заголовках, а не о телах.
   а. Функция n_to_char() принимает аргумент int и возвращает значение типа char.
   б. Функция digits() принимает аргументы double и int и возвращает значение типа int.
   в. Функция which() принимает в качестве аргументов два адреса значений double и возвращает адрес значения типа double.
   г. Функция random() не принимает аргументов и возвращает значение типа int.
4. Напишите функцию, которая возвращает сумму двух целых чисел.
5. Что придется изменить, если вообще придется, чтобы функция из вопроса 4 взамен суммировала два числа типа double?
6. Напишите функцию по имени alter(), которая принимает две переменные int, x и y, и устанавливает их значения, соответственно, в сумму и разность x и y.
7. Нет ли ошибок в следующем определении функции?

```
void salami(num)
{
   int num, count;

   for (count = 1; count <= num; num++)
      printf(" Здравствуйте!\n");
}
```

8. Напишите функцию, которая возвращает наибольший из трех целочисленных аргументов.
9. Задан следующий вывод:

```
Выберите один из следующих вариантов:
1) копировать файлы       2) переместить файлы
3) удалить файлы          4) выйти из программы
Введите номер выбранного варианта:
```

   а. Напишите функцию, которая выводит на экран меню из четырех пронумерованных вариантов и предлагает выбрать один из них. (Вывод должен иметь показанный выше вид.)

б. Напишите функцию, которая имеет два аргумента int: нижний предел и верхний предел. Функция должна читать целое число из входных данных. Если это число выходит за указанные пределы, функция должна снова вывести меню (используя функцию из части а)), чтобы повторно предложить пользователю ввести новое значение. Если введенное целое значение попадает в рамки пределов, функция должна возвратить его в вызывающую функцию. Ввод нецелочисленного значения должен приводить к возвращению функцией значения, соответствующего выходу из программы (4).

в. Напишите минимальную программу, применяя функции из частей а) и б) этого вопроса. Под *минимальной* понимается то, что она не должна действительно выполнять действия, объявленные в меню; вы должны только отобразить варианты и получить допустимый ответ.

Упражнения по программированию

1. Напишите функцию по имени min(x,y), которая возвращает меньшее из двух значений double. Протестируйте эту функцию с помощью простого драйвера.

2. Напишите функцию по имени chline(ch,i,j), которая выводит требуемый символ в столбцах с i по j. Протестируйте эту функцию с помощью простого драйвера.

3. Напишите функцию, которая принимает три аргумента: символ и два целых числа. Символ предназначен для вывода. Первое целое значение задает количество указанных символов в строке, а второе целое число устанавливает количество таких строк. Напишите программу, в которой используется эта функция.

4. Среднее гармоническое значение двух чисел получается путем инвертирования этих чисел, вычисления среднего значения инверсий и получения инверсии результата. Напишите функцию, которая принимает два аргумента double и возвращает среднее гармоническое значение этих двух чисел.

5. Напишите и протестируйте функцию по имени larger_of(), которая заменяет содержимое двух переменных double большим из их значений. Например, вызов larger_of(x,y) присвоит переменным x и y большее из их значений.

6. Напишите и протестируйте функцию, которая принимает в качестве аргументов адреса трех переменных double и помещает наименьшее значение в первую переменную, среднее значение — во вторую, а наибольшее значение — в третью.

7. Напишите программу, которая читает символы из стандартного ввода вплоть до конца файла. Для каждого символа программа должна сообщать, является ли он буквой. Если символ — буква, программа вдобавок должна сообщать ее порядковый номер в алфавите. Например, буквы *c* и *C* будут иметь номер 3. Предусмотрите в программе функцию, которая принимает символ в качестве аргумента и возвращает его порядковый номер в алфавите, если он является буквой, и -1 в противном случае.

8. В главе 6 была показана функция power() (листинг 6.20), которая возвращает результат возведения числа double в положительную целую степень. Усовершенствуйте эту функцию, чтобы она корректно возводила числа в отрицательные степени. Кроме того, добавьте в функцию возможность оценки как 0 результата возведения 0 в любую степень кроме 0 и оценки как 1 результата

возведения любого числа в степень 0. (Функция должна сообщать, что результат возведения 0 в степень 0 не определен и что она использует значение 1.) Примените цикл. Протестируйте функцию в какой-нибудь программе.

9. Еще раз выполните упражнение 8, но на этот раз используйте рекурсивную функцию.

10. Обобщите функцию `to_binary()` из листинга 9.8 до функции `to_base_n()`, которая принимает второй аргумент в диапазоне от 2 до 10. Она должна выводить число, переданное в первом аргументе, в системе счисления с основанием, которое указано во втором аргументе. Например, вызов `to_base_n(129,8)` должен отобразить `201`, т.е. восьмеричный эквивалент числа `129`. Протестируйте готовую функцию в какой-нибудь программе.

11. Напишите и протестируйте функцию `Fibonacci()`, в которой для вычисления чисел Фибоначчи вместо рекурсии применяется цикл.

10

Массивы и указатели

В ЭТОЙ ГЛАВЕ...

- Ключевое слово: static
- Операции: &* (унарная)
- Создание и инициализация массивов
- Указатели (на основе сведений, которые вам уже известны) и их отношение к массивам
- Написание функций, обрабатывающих массивы
- Двумерные массивы

Люди обращаются к компьютерам для решения таких задач, как отслеживание ежемесячных расходов, ежедневного количества осадков, ежеквартальных продаж и еженедельного веса. Предприятия применяют компьютеры для управления платежными ведомостями, складом и транзакциями от заказчиков. Будучи программистом, вы неизбежно вынуждены иметь дело с большими объемами связанных данных. Часто массивы предлагают наилучший способ обработки таких данных в эффективной и удобной манере. Вводные сведения о массивах были представлены в главе 6, а в этой главе массивы рассматриваются более подробно. В частности, мы исследуем приемы написания функций для обработки массивов. Такие функции позволяют распространить на массивы преимущества модульного программирования. По ходу дела вы сможете увидеть тесную связь между массивами и указателями.

Массивы

Вспомните, что *массив* состоит из последовательности элементов одного типа данных. Для сообщения компилятору о том, что нужен массив, используется *объявление*. В *объявлении массива* указывается, сколько элементов содержит массив и какого типа эти элементы. Располагая такой информацией, компилятор может подходящим образом создать массив. Элементы массива могут иметь те же самые типы, что и обычные переменные. Рассмотрим следующие примеры объявлений массивов:

```
/* несколько объявлений массивов */
int main(void)
{
    float candy[365];     /* массив из 365 элементов типа float */
    char code[12];        /* массив из 12 элементов типа char   */
    int states[50];       /* массив из 50 элементов типа int    */
   ...
}
```

Квадратные скобки ([]) идентифицируют candy и другие имена в качестве массивов, а число в квадратных скобках задает количество элементов в массиве.

При доступе к элементам в массиве вы указываете отдельный элемент с применением его номера, который также называется *индексом*. Нумерация элементов начинается с 0. Следовательно, candy[0] — это первый элемент массива candy, а candy[364] — 365-й, и последний, элемент массива.

Это довольно хорошо знакомо, так что давайте ознакомимся с чем-нибудь новым.

Инициализация

Массивы часто используются для хранения данных, необходимых для программы. Например, 12-элементный массив может хранить количество дней в каждом месяце. В случаях подобного рода удобно инициализировать массив в начале программы. Посмотрим, как это делается.

Вы знаете, как можно инициализировать однозначные переменные (иногда называемые *скалярными*) в объявлениях с помощью таких выражений, как показанные ниже:

```
int fix = 1;
float flax = PI * 2;
```

Здесь предполагается, что макрос PI был определен ранее. Язык С расширяет инициализацию на массивы посредством нового синтаксиса:

```
int main(void)
{
   int powers[8] = {1,2,4,6,8,16,32,64}; /* ANSI C и последующие стандарты */
   ...
}
```

Нетрудно заметить, что массив инициализируется с применением списка значений, разделяемых запятыми, который заключен в квадратные скобки. При желании между запятыми и значениями можно помещать пробелы. Первому элементу (powers[0]) присваивается значение 1, второму (powers[1]) – значение 2 и т.д. (Если ваш компилятор отклоняет такую форму инициализации как синтаксически некорректную, значит, он был разработан до выхода стандарта ANSI. Проблему должно решить помещение перед объявлением массива ключевого слова static, которое более подробно обсуждается в главе 12.)

В листинге 10.1 приведена короткая программа, которая выводит количество дней в каждом месяце.

Листинг 10.1. Программа day_mon1.c

```
/* day_mon1.c -- выводит количество дней в каждом месяце */
#include <stdio.h>
#define MONTHS 12

int main(void)
{
   int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
   int index;

   for (index = 0; index < MONTHS; index++)
      printf("Месяц %d имеет %2d день (дней).\n", index +1,
             days[index]);

   return 0;
}
```

Вывод выглядит следующим образом:

```
Месяц 1 имеет 31 день (дней).
Месяц 2 имеет 28 день (дней).
Месяц 3 имеет 31 день (дней).
Месяц 4 имеет 30 день (дней).
Месяц 5 имеет 31 день (дней).
Месяц 6 имеет 30 день (дней).
Месяц 7 имеет 31 день (дней).
Месяц 8 имеет 31 день (дней).
Месяц 9 имеет 30 день (дней).
Месяц 10 имеет 31 день (дней).
Месяц 11 имеет 31 день (дней).
Месяц 12 имеет 31 день (дней).
```

Не особо впечатляющая программа, но она сообщает некорректные сведения только для одного месяца раз в четыре года. Программа инициализирует массив days[] списком разделенных запятыми значений, заключенным в квадратные скобки.

Обратите внимание, что в этом примере для представления размера массива используется символическая константа MONTHS. Это распространенная и рекомендованная практика. Например, если вдруг мир перейдет на 13-месячный календарь, понадобится только модифицировать оператор #define, но не отслеживать все места в программе, где задействован размер массива.

НА ЗАМЕТКУ! Использование констант в массивах

Иногда приходится применять массив, предназначенный только для чтения. То есть программа будет извлекать из него значения, но не пытаться записывать новые значения в этот массив. В подобных случаях вы можете, да и должны, использовать ключевое слово const во время объявления и инициализации массива. Таким образом, в листинге 10.1 лучше указать следующее объявление:

```
const int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Это заставляет программу трактовать каждый элемент массива как константу. Как и в случае обычных переменных, для инициализации данных const вы должны применять объявление, поскольку из-за наличия const присвоить им значения позже не получится. Теперь, когда это известно, мы можем использовать константы в последующих примерах.

А что, если вы забудете инициализировать массив? В листинге 10.2 показано, что произойдет.

Листинг 10.2. Программа no_data.c

```
/* no_data.c -- неинициализированный массив */
#include <stdio.h>
#define SIZE 4
int main(void)
{
   int no_data[SIZE];  /* неинициализированный массив */
   int i;
   printf("%2s%14s\n",
          "i", "no_data[i]");
   for (i = 0; i < SIZE; i++)
      printf("%2d%14d\n", i, no_data[i]);
   return 0;
}
```

Ниже приведен результирующий вывод одного пробного запуска (ваши результаты могут отличаться):

```
 i    no_data[i]
 0             0
 1       4204937
 2       4219854
 3    2147348480
```

Элементы массива похожи на обычные переменные – если вы не инициализируете их, они могут иметь любые значения. Компилятору разрешено просто брать значения, которые уже находятся в соответствующих ячейках памяти; именно по этой причине ваши результаты могут отличаться от показанных выше.

НА ЗАМЕТКУ! Пояснение по поводу классов хранения

Массивы, как и другие переменные, могут быть созданы с применением различных *классов хранения*. Эта тема исследуется в главе 12, но пока достаточно знать, что в текущей главе описаны массивы, которые принадлежат к автоматическому классу хранения. Это означает, что они объявлены внутри функции без указания ключевого слова static. Все переменные и массивы, использованные до сих пор в книге, относятся к автоматическому классу хранения. Причина упоминания здесь классов хранения заключается в том, что иногда разные классы хранения обладают отличающимися свойствами, поэтому вы не должны распространять все сказанное в настоящей главе на другие классы хранения. В частности, переменные и массивы с некоторыми другими классами хранения, не будучи инициализированными, имеют содержимое, установленное в 0.

Количество элементов в списке должно соответствовать размеру массива. Но что, если вы подсчитали неправильно? Давайте возвратимся к последнему примеру, как показано в листинге 10.3, сократив список инициализации до двух элементов.

Листинг 10.3. Программа somedata.c

```
/* some_data.c -- частично инициализированный массив */
#include <stdio.h>
#define SIZE 4
int main(void)
{
   int some_data[SIZE] = {1492, 1066};
   int i;

   printf("%2s%14s\n",
         "i", "some_data[i]");
   for (i = 0; i < SIZE; i++)
      printf("%2d%14d\n", i, some_data[i]);

   return 0;
}
```

На этот вывод выглядит следующим образом:

```
i  some_data[i]
0          1492
1          1066
2             0
3             0
```

Как видите, у компилятора не возникло никаких проблем. Когда значения в списке закончились, он инициализировал остальные элементы значением 0. Другими словами, если вы вообще не инициализируете массив, то его элементы, подобно обычным переменным, получают случайные значения из памяти, но если вы инициализируете массив частично, то оставшиеся элементы устанавливаются в 0.

Однако компилятор не настолько великодушен, если список содержит слишком много значений. Такая избыточность считается ошибкой. Тем не менее, нет никакой необходимости подвергать себя насмешкам со стороны вашего компилятора. Вместо этого вы можете позволить компилятору привести размер массива в соответствие со списком, ничего не указывая в квадратных скобках (листинг 10.4).

Листинг 10.4. Программа day_mon2.c

```
/* day_mon2.c -- предоставление компилятору возможности подсчета элементов */
#include <stdio.h>
int main(void)
{
   const int days[] = {31,28,31,30,31,30,31,31,30,31};
   int index;

   for (index = 0; index < sizeof days / sizeof days[0]; index++)
      printf("Месяц %2d имеет %d день (дней).\n", index +1,
            days[index]);

   return 0;
}
```

В листинге 10.4 необходимо отметить два основных момента.

- Когда вы применяете для инициализации массива пустые квадратные скобки, компилятор подсчитывает количество элементов в списке и устанавливает размер массива в полученное число.
- Обратите внимание на то, что мы делали в управляющем операторе цикла `for`. Из-за отсутствия уверенности в возможности корректного подсчета количества элементов мы позволили компьютеру самостоятельно определить размер массива. Операция `sizeof` выдает размер в байтах следующего за ней объекта или *типа*. Таким образом, `sizeof days` — это размер в байтах всего массива, а `sizeof days[0]` — размер в байтах одного элемента. Разделив размер всего массива на размер одного элемента, мы получаем количество элементов в массиве.

Ниже показан результат выполнения этой программы:

```
Месяц 1 имеет 31 день (дней).
Месяц 2 имеет 28 день (дней).
Месяц 3 имеет 31 день (дней).
Месяц 4 имеет 30 день (дней).
Месяц 5 имеет 31 день (дней).
Месяц 6 имеет 30 день (дней).
Месяц 7 имеет 31 день (дней).
Месяц 8 имеет 31 день (дней).
Месяц 9 имеет 30 день (дней).
Месяц 10 имеет 31 день (дней).
```

Вот как! Мы указали только 10 значений, но метод с предоставлением программе возможности самостоятельно определить размер массива предотвратил попытку вывода значений за пределами массива. Это подчеркивает потенциальный недостаток автоматического подсчета: ошибки в количестве элементов могут пройти незамеченными.

Существует еще один более короткий метод инициализации массивов. Однако, поскольку он работает только для символьных строк, мы отложим его рассмотрение до следующей главы.

Назначенные инициализаторы (C99)

В стандарте C99 добавлена новая возможность — *назначенные инициализаторы*. Это средство позволяет выбирать, какие элементы будут инициализированы. Предположим, например, что вы хотите инициализировать только последний элемент в массиве. С помощью традиционного синтаксиса инициализации языка С понадобится также инициализировать все элементы, предшествующие последнему:

```
int arr[6] = {0,0,0,0,0,212}; // традиционный синтаксис
```

Стандарт C99 позволяет применять в списке инициализации индекс в квадратных скобках, чтобы указать конкретный элемент:

```
int arr[6] = {[5] = 212};      // инициализация элемента arr[5] значением 212
```

Как и при обычной инициализации, после того, как вы инициализируете хотя бы один элемент, оставшиеся неинициализированные элементы устанавливаются в 0. В листинге 10.5 представлен более сложный пример.

Листинг 10.5. Программа designate.c

```
// designate.c -- использование назначенных инициализаторов
#include <stdio.h>
#define MONTHS 12
int main(void)
{
   int days[MONTHS] = {31,28, [4] = 31,30,31, [1] = 29};
   int i;

   for (i = 0; i < MONTHS; i++)
      printf("%2d %d\n", i + 1, days[i]);

   return 0;
}
```

Вот вывод для случая, когда компилятор поддерживает это средство C99:

```
 1  31
 2  29
 3  0
 4  0
 5  31
 6  30
 7  31
 8  0
 9  0
10  0
11  0
12  0
```

Вывод отражает несколько важных характеристик назначенных инициализаторов. Во-первых, если за назначенным инициализатором находится код с дальнейшими значениями, как в последовательности `[4] = 31,30,31`, эти значения используются для инициализации последующих элементов. То есть после инициализации `days[4]` значением `31` этот код инициализирует `days[5]` и `days[6]` значениями `30` и `31`, соответственно. Во-вторых, если код инициализирует отдельный элемент значением более одного раза, то актуальной будет последняя инициализация. Например, в листинге 10.5 в начале списка производится инициализация `days[1]` значением `28`, но позже это значение переопределено назначенным инициализатором `[1] = 29`.

А что произойдет, если вы не указали размер массива?

```
int stuff[] = {1, [6] = 23};        // что происходит?
int staff[] = {1, [6] = 4, 9, 10};  // что происходит?
```

Компилятор сделает массив достаточно большим, чтобы уместить значения инициализации. Так, массив `stuff` будет иметь семь элементов с номерами 0–6, а массив `staff` — на два элемента больше, т.е. 9 элементов.

Присваивание значений элементам массива

После того, как массив был объявлен, элементам массива можно *присваивать* значения с применением их *индексов*. Например, в следующем фрагменте элементам массива присваиваются четные числа:

```
/* присваивание значений элементам массива */
#include <stdio.h>
#define SIZE 50
```

```
int main(void)
{
   int counter, evens[SIZE];

   for (counter = 0; counter < SIZE; counter++)
      evens[counter] = 2 * counter;
   ...
}
```

Обратите внимание, что в коде используется цикл для поэлементного присваивания значений. Язык C не позволяет присваивать один массив другому как единый модуль. Кроме того, нельзя применять форму списка в фигурных скобках, кроме как при инициализации. В следующем фрагменте кода показаны некоторые недопустимые формы присваивания.

```
/* недопустимые формы присваивания значений элементам массива */
#define SIZE 5
int main(void)
{
   int oxen[SIZE] = {5,3,2,8};     /* здесь все в порядке          */
   int yaks[SIZE];

   yaks = oxen;                    /* не разрешено                 */
   yaks[SIZE] = oxen[SIZE];        /* выход за пределы диапазона   */
   yaks[SIZE] = {5,3,2,8};         /* не работает                  */
```

Вспомните, что последним элементом массива oxen является oxen[SIZE-1], поэтому oxen[SIZE] и yaks[SIZE] ссылаются на данные, находящиеся за последними элементами обоих массивов.

Границы массива

Вы должны обеспечить, чтобы используемые индексы массива не выходили за границы, т.е. удостовериться в том, что они имеют значения, допустимые для массива. Например, предположим, что имеется следующее объявление:

```
int doofi[20];
```

После этого на вас возлагается ответственность на то, что в программе будут применяться только индексы из диапазона от 0 до 19, т.к. компилятор не обязан вас проверять. (Хотя некоторые компиляторы будут предупреждать о наличии проблемы, но продолжат компиляцию программы в любом случае.)

Рассмотрим программу из листинга 10.6. Она создает массив с четырьмя элементами, а затем беспечно использует значения индекса в диапазоне от -1 до 6.

Листинг 10.6. Программа bounds.c

```
// bounds.c -- выход за границы массива
#include <stdio.h>
#define SIZE 4
int main(void)
{
   int value1 = 44;
   int arr[SIZE];
   int value2 = 88;
   int i;

   printf("value1 = %d, value2 = %d\n", value1, value2);
   for (i = -1; i <= SIZE; i++)
      arr[i] = 2 * i + 1;
```

```
    for (i = -1; i < 7; i++)
       printf("%2d %d\n", i, arr[i]);
    printf("value1 = %d, value2 = %d\n", value1, value2);
    printf("адрес arr[-1]: %p\n", &arr[-1]);
    printf("адрес arr[4]: %p\n", &arr[4]);
    printf("адрес value1: %p\n", &value1);
    printf("адрес value2: %p\n", &value2);
    return 0;
}
```

Компилятор не проверяет допустимость индексов. В стандарте языка С результат применения некорректного индекса является неопределенным. Это означает, что после запуска программа может выглядеть работоспособной, функционировать странным образом или вовсе аварийно завершиться. Ниже приведен пример вывода при использовании компилятора GCC:

```
value1 = 44, value2 = 88
-1 -1
 0 1
 1 3
 2 5
 3 7
 4 9
 5 1624678494
 6 32767
value1 = 9, value2 = -1
адрес arr[-1]: 0x7fff5fbff8cc
адрес arr[4]: 0x7fff5fbff8e0
адрес value1: 0x7fff5fbff8e0
адрес value2: 0x7fff5fbff8cc
```

Обратите внимание, что компилятор сохранил значение value1 непосредственно после массива, а значение value2 — прямо перед ним. (Другие компиляторы могут сохранять данные в память в другом порядке.) В этом случае, как показано в выводе, arr[-1] соответствует той же ячейке памяти, что и value2, а arr[4] — той же ячейке памяти, что и value1. Следовательно, применение индексов, выходящих за границы массива, приводит к тому, что программа изменяет значения других переменных. Другой компилятор может дать другие результаты, включая аварийное завершение программы.

Может возникнуть вопрос, почему в С подобное разрешено. Это является следствием принятой в языке философии доверия программисту. Отсутствие проверки границ позволяет программе на С выполняться быстрее. Компилятор не всегда способен выявить все ошибки индексации, т.к. значение индекса может оставаться неопределенным до тех пор, пока не начнется выполнение программы. По этой причине для обеспечения безопасности компилятору пришлось бы добавлять дополнительный код для проверки каждого индекса во время выполнения, что приводило бы к снижению скорости выполнения. Таким образом, компилятор С доверяет программисту в том, что он корректно кодирует, и вознаграждает его более быстрой программой. Конечно, не все программисты заслуживают такого доверия, и в таких случаях могут возникать проблемы.

Запомните одну простую вещь: нумерация в массиве начинается с 0. Необходимо выработать привычку использовать символическую константу в объявлении массива и в других местах, где применяется размер массива:

```
#define SIZE 4
int main(void)
{
   int arr[SIZE];
   for (i = 0; i < SIZE; i++)
   ....
```

Это поможет обеспечить согласованное использование размера массива повсеместно в программе.

Указание размера массива

До сих пор при объявлении массивов применялись целочисленные константы:

```
#define SIZE 4
int main(void)
{
   int arr[SIZE];          // символическая целочисленная константа
   double lots[144];       // литеральная целочисленная константа
   ...
```

Что еще разрешено? До выхода стандарта C99 при объявлении массива в квадратных скобках вы должны были помещать *константное целочисленное выражение* — выражение, сформированное из целочисленных констант. В этом смысле выражение sizeof считается целочисленной константой, но (в отличие от такого случая в C++) значение const — нет. Кроме того, значение такого выражения должно было быть больше 0:

```
int n = 5;
int m = 8;
float a1[5];                 // да
float a2[5*2 + 1];           // да
float a3[sizeof(int) + 1];   // да
float a4[-4];                // нет, размер должен быть > 0
float a5[0];                 // нет, размер должен быть > 0
float a6[2.5];               // нет, размер должен быть целым числом
float a7[(int)2.5];          // да, приведение константы float к типу int
float a8[n];                 // не было разрешено до появления стандарта C99
float a9[m];                 // не было разрешено до появления стандарта C99
```

Как показывают комментарии, компиляторы C, соответствующие стандарту C90, не разрешают два последних объявления. Однако, начиная со стандарта C99, в языке они допускаются, но приводят к созданию нового вида массивов, которые называются *массивами переменной длины*. (В стандарте C11 отступили от этой смелой инициативы, сделав массивы переменной длины дополнительной, а не обязательной языковой возможностью.)

Массивы переменной длины были введены в стандарт C99 главным образом для того, чтобы дать возможность C стать лучшим языком в плане числовых вычислений. Например, массивы переменной длины облегчают преобразование существующих библиотек подпрограмм цифровых расчетов на языке FORTRAN в код C. Массивы переменной длины обладают рядом ограничений; к примеру, массив переменной длины нельзя инициализировать при его объявлении. В этой главе мы еще вернемся к массивам переменной длины после того, как вы изучите ограничения классического массива C.

Многомерные массивы

Мисс Темпест Клауд, метеоролог, желает проанализировать данные об осадках за последние пять лет. Первым делом ей необходимо выбрать способ представления данных. Один из вариантов предусматривает использование 60 переменных, по одной для каждого элемента данных. (Ранее мы уже упоминали этот вариант; сейчас, как и тогда, в нем мало смысла.) Массив из 60 элементов представляется более совершенным способом, но намного лучше хранить данные для каждого года отдельно. Можно было бы применять 5 массивов по 12 элементов, но это грубый подход, который превратится в трудноразрешимую проблему, если мисс Клауд решит изучить данные об осадках за 50 лет вместо пяти. Словом, ей нужно найти что-нибудь получше.

Более эффективный подход предполагает использование массива массивов. Главный массив должен иметь пять элементов, по одному на каждый год. В свою очередь, каждый из этих элементов является 12-элементным массивом, по одному элементу на каждый месяц. Такой массив объявляется следующим образом:

```
float rain[5][12]; // массив из 5 массивов по 12 элементов float
```

Можно взглянуть сначала на внутреннюю часть приведенного объявления, которая выделена полужирным:

```
float rain[5][12]; // rain - массив, содержащий пять пока невыясненных сущностей
```

Внутренняя часть говорит о том, что `rain` — это массив с пятью элементами. Но что представляет собой каждый из этих элементов? Теперь обратимся к оставшейся части объявления (снова выделенной полужирным):

```
float rain[5][12]; // массив из 12 значений float
```

Это информирует о том, что каждый элемент имеет тип `float[12]`, т.е. каждый из пяти элементов `rain` сам по себе является массивом из 12 значений `float`.

Согласно такой логике, `rain[0]`, будучи первым элементом массива `rain`, представляет собой массив из 12 значений `float`. То же самое касается `rain[1]`, `rain[2]` и т.д. Если `rain[0]` представляет собой массив, то его первым элементом будет `rain[0][0]`, вторым элементом — `rain[0][1]` и т.д. Короче говоря, `rain` — это 5-элементный массив из 12-элементных массивов `float`, `rain[0]` — массив из 12 элементов `float`, а `rain[0][0]` — значение `float`. Для доступа, скажем, к значению в строке 2 и столбце 3 применяется запись `rain[2][3]`. (Не забывайте, что отсчет начинается с 0, поэтому строка с номером 2 будет физически третьей.)

Массив `rain` можно представить в виде двумерного массива, состоящего из пяти строк, каждая из которых содержит 12 столбцов (рис. 10.1). Изменяя второй индекс, вы перемещаетесь по строке, месяц за месяцем. Изменяя первый индекс, вы переходите вертикально вдоль столбца, год за годом.

Двумерное представление — это всего лишь удобный способ визуализации массива с двумя индексами. Внутренне такой массив хранится последовательно, начиная с первого 12-элементного массива, за которым следует второй 12-элементный массив, и т.д.

Давайте воспользуемся этим двумерным массивом в программе обработки погодных данных. Цель программы заключается в нахождении итоговой суммы осадков для каждого года, средних значений осадков за год и средних значений осадков за месяц. Чтобы вычислить итоговую сумму осадков за год, необходимо сложить все данные в отдельной строке. Чтобы получить среднее значение осадков за конкретный месяц, понадобится сложить все значения в заданном столбце. Двумерный массив упрощает визуальное представление и выполнение этих действий. Программа приведена в листинге 10.7.

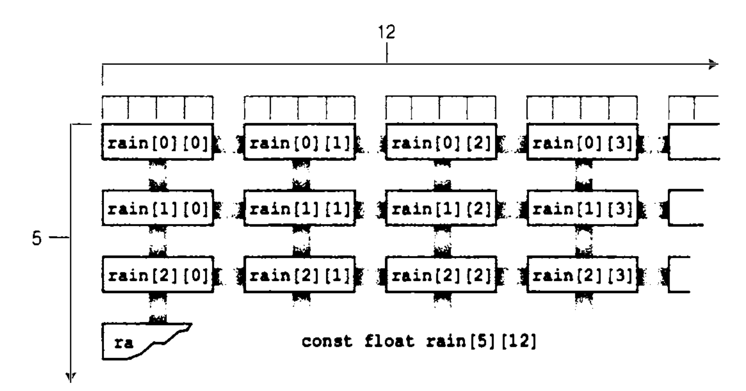


Рис. 10.1. Двумерный массив

Листинг 10.7. Программа rain.c

```
/* rain.c -- находит суммарные данные по годам, средние значения за год и средние
            значения за месяц по данным об осадках за несколько лет */
#include <stdio.h>
#define MONTHS 12   // количество месяцев в году
#define YEARS 5     // количество лет, для которых доступны данные
int main(void)
{
    // инициализация данными об осадках за период с 2010 по 2014 гг.
    const float rain[YEARS][MONTHS] =
    {
        {4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6},
        {8.5,8.2,1.2,1.6,2.4,0.0,5.2,0.9,0.3,0.9,1.4,7.3},
        {9.1,8.5,6.7,4.3,2.1,0.8,0.2,0.2,1.1,2.3,6.1,8.4},
        {7.2,9.9,8.4,3.3,1.2,0.8,0.4,0.0,0.6,1.7,4.3,6.2},
        {7.6,5.6,3.8,2.8,3.8,0.2,0.0,0.0,0.0,1.3,2.6,5.2}
    };
    int year, month;
    float subtot, total;
    printf("ГОД    КОЛИЧЕСТВО ОСАДКОВ (в дюймах)\n");
    for (year = 0, total = 0; year < YEARS; year++)
    {   // для каждого года суммировать количество осадков за каждый месяц
        for (month = 0, subtot = 0; month < MONTHS; month++)
            subtot += rain[year][month];
        printf("%5d %15.1f\n", 2010 + year, subtot);
        total += subtot; // общая сумма для всех лет
    }
    printf("\nСреднегодовое количество осадков составляет %.1f дюймов.\n\n",
            total/YEARS);
    printf("СРЕДНЕМЕСЯЧНОЕ КОЛИЧЕСТВО ОСАДКОВ:\n\n");
    printf(" Янв  Фев  Мар  Апр  Май  Июн  Июл  Авг  Сен  Окт");
    printf("  Ноя  Дек\n");
    for (month = 0; month < MONTHS; month++)
    {   // для каждого месяца суммировать количество осадков на протяжении годов
        for (year = 0, subtot = 0; year < YEARS; year++)
            subtot += rain[year][month];
        printf("%4.1f ", subtot/YEARS);
    }
    printf("\n");
    return 0;
}
```

Ниже показан вывод программы:

```
 ГОД   КОЛИЧЕСТВО ОСАДКОВ (в дюймах)
2010          32.4
2011          37.9
2012          49.8
2013          44.0
2014          32.9

Среднегодовое количество осадков составляет 39.4 дюймов.

СРЕДНЕМЕСЯЧНОЕ КОЛИЧЕСТВО ОСАДКОВ:

Янв  Фев  Мар  Апр  Май  Июн  Июл  Авг  Сен  Окт  Ноя  Дек
7.3  7.3  4.9  3.0  2.3  0.6  1.2  0.3  0.5  1.7  3.6  6.7
```

При изучении этой программы сосредоточьте внимание на инициализации и на схеме вычислений. Из этих двух частей инициализация является более сложной, так что сначала давайте рассмотрим часть, которая проще (вычисления).

Чтобы найти итоговую сумму для заданного года, оставьте year неизменным и позвольте month пройти через весь диапазон значений. Это реализует внутренний цикл for в первой части программы. Затем повторите процесс для следующего значения year. Именно это делает внешний цикл в первой части программы. Структура с вложенными циклами подобного рода является естественной для обработки двумерного массива. Один цикл обрабатывает первый индекс, а второй цикл — второй индекс:

```
for (year = 0, total = 0; year < YEARS; year++)
{               // обработка каждого года
   for (month = 0, subtot = 0; month < MONTHS; month++)
      ...       // обработка каждого месяца
   ...          // обработка каждого года
}
```

Вторая часть программы имеет ту же самую структуру, но теперь year изменяется во внутреннем цикле, а month — во внешнем. Вспомните, что каждый раз, когда внешний цикл выполняет одну итерацию, внутренний цикл проходит через все свои итерации. Следовательно, при такой организации, прежде чем изменится месяц, цикл просматривает все года. Мы получаем среднее значение за пятилетний период для первого месяца, для второго месяца и т.д.

```
for (month = 0; month < MONTHS; month++)
{               // обработка каждого месяца
   for (year = 0, subtot =0; year < YEARS; year++)
      ...       // обработка каждого года
   ...          // обработка каждого месяца
}
```

Инициализация двумерного массива

Инициализация двумерного массива построена на приеме, применяемом для инициализации одномерного массива. Прежде всего, вспомните, что инициализация одномерного массива выглядит следующим образом:

```
sometype ar1[5] = {val1, val2, val3, val4, val5};
```

Здесь val1, val2 и т.д. являются значениями типа sometype. Например, если бы типом sometype был int, то значением val1 могло быть 7, а если бы типом sometype был double, то значением val1 могло быть 11.34. Но rain — это 5-элементный массив, каждый элемент которого является массивом, состоящим из 12 значений float.

Следовательно, для rain в качестве val1 должно быть значение, пригодное для инициализации одномерного массива значений float, такое как:

```
{4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6}
```

Другими словами, если sometype – массив из 12 значений double, то val1 – это список из 12 значений double. Таким образом, для инициализации двумерного массива, подобного rain, нам необходим список из пяти таких сущностей, разделенных запятыми:

```
const float rain[YEARS][MONTHS] =
{
    {4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6},
    {8.5,8.2,1.2,1.6,2.4,0.0,5.2,0.9,0.3,0.9,1.4,7.3},
    {9.1,8.5,6.7,4.3,2.1,0.8,0.2,0.2,1.1,2.3,6.1,8.4},
    {7.2,9.9,8.4,3.3,1.2,0.8,0.4,0.0,0.6,1.7,4.3,6.2},
    {7.6,5.6,3.8,2.8,3.8,0.2,0.0,0.0,0.0,1.3,2.6,5.2}
};
```

В этой инициализации используются пять заключенных в фигурные скобки списков чисел, которые помещены во внешний набор фигурных скобок. Данные из первой внутренней пары фигурных скобок присваиваются первой строке массива, данные из второй внутренней пары скобок – второй строке массива и т.д. Рассмотренные выше правила относительно несоответствий между количеством данных и размером массива, применяются к каждой строке. Это значит, что если внутренний набор фигурных скобок содержит 10 чисел, то воздействие будет оказано только на начальные 10 элементов в первой строке. Последние два элемента в этой строке по умолчанию инициализируются нулем. Если чисел задано слишком много, возникает ошибка; числа не переносятся в следующую строку.

Внутренние фигурные скобки можно было бы не указывать, оставив только две внешних скобки. При правильном количестве записей результат будет таким же. Однако если записей недостаточно, массив заполняется последовательно, строка за строкой, пока данные не закончатся. Затем оставшиеся элементы инициализируются значением 0. На рис. 10.2 продемонстрированы оба способа инициализации массива.

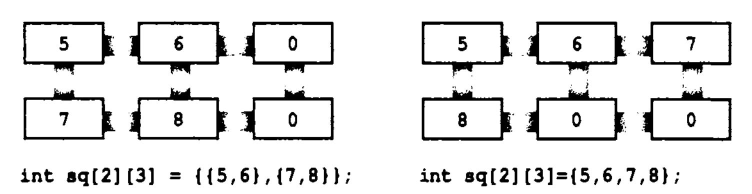


Рис. 10.2. Два метода инициализации массива

Поскольку массив rain содержит данные, которые не должны быть модифицированы, при объявлении массива в программе используется модификатор const.

Большее количество измерений

Все, что было сказано о двумерных массивах, можно распространить на трехмерные массивы и на массивы с большим числом измерений. Трехмерный массив объявляется следующим образом:

```
int box[10][20][30];
```

Одномерный массив можно представлять себе как строку данных, двумерный массив – как таблицу данных, а трехмерный массив – как стопку таблиц данных.

Например, о массиве box можно думать как о 10 двумерных массивах (каждый размером 20×30), уложенных друг на друга.

Массив box можно по-другому представлять себе как массив массивов, состоящих из массивов. То есть это 10-элементный массив, элементами которого являются 20-элементные массивы. Каждый 20-элементный массив содержит элементы, представляющие собой 30-элементные массивы. Либо же массивы можно просто рассматривать с точки зрения количества необходимых индексов.

Обычно для обработки трехмерного массива применяются три вложенных цикла, для обработки четырехмерного массива – четыре вложенных цикла и т.д. В своих примерах мы ограничимся двумерными массивами.

Указатели и массивы

Как объяснялось в главе 9, указатели предоставляют символический способ работы с адресами. Поскольку аппаратные инструкции вычислительных машин в большой степени полагаются на адреса, указатели позволяют выражать действия в манере, близкой к машинному представлению. Такое соответствие делает программы с указателями эффективными. В частности, указатели предлагают эффективный метод манипулирования массивами. На самом деле, как вы увидите далее, система обозначения массивов является просто замаскированным использованием указателей.

Примером такого замаскированного применения может служить тот факт, что имя массива представляет собой также и адрес его первого элемента. Это означает, что если flizny – массив, то следующее выражение будет истинным:

```
flizny == &flizny[0];  // имя массива является адресом его первого элемента
```

И flizny, и &flizny[0] представляют адрес в памяти, где находится первый элемент массива. (Вспомните, что & – операция взятия адреса.) Кроме того, это *константы*, т.к. они остаются фиксированными на протяжении всего времени действия программы. Тем не менее, их можно присваивать в качестве значений *переменной* типа указателя, значение которой можно изменять, как показано в листинге 10.8. Посмотрите, что происходит со значением указателя, когда вы прибавляете к нему число. (Как вы, возможно, помните, спецификатор %p для указателей обычно приводит к отображению их шестнадцатеричных значений.)

Листинг 10.8. Программа pnt_add.c

```
// pnt_add.c -- сложение указателей
#include <stdio.h>
#define SIZE 4
int main(void)
{
   short dates [SIZE];
   short * pti;
   short index;
   double bills[SIZE];
   double * ptf;
   pti = dates;    // присваивание указателю адреса массива
   ptf = bills;
   printf("%23s %15s\n", "short", "double");
   for (index = 0; index < SIZE; index ++)
      printf("указатели + %d: %10p %10p\n",
             index, pti + index, ptf + index);
   return 0;
}
```

Вот пример вывода:

```
                         short          double
указатели + 0: 0x7fff5fbff8dc 0x7fff5fbff8a0
указатели + 1: 0x7fff5fbff8de 0x7fff5fbff8a8
указатели + 2: 0x7fff5fbff8e0 0x7fff5fbff8b0
указатели + 3: 0x7fff5fbff8e2 0x7fff5fbff8b8
```

Во второй строке выводятся начальные адреса двух массивов, в следующей после нее строке показан результат прибавления к адресу 1 и т.д. Имейте в виду, что адреса представлены в шестнадцатеричной форме, поэтому dd на 1 больше, чем dc, а a1 на 1 больше, чем a0. Что же мы здесь имеем?

```
0x7fff5fbff8dc + 1 = 0x7fff5fbff8de?
0x7fff5fbff8a0 + 1 = 0x7fff5fbff8a8?
```

Довольно глупо? Нет – хитро! В нашей системе реализована побайтная адресация, но тип short занимает 2 байта, а тип double – 8 байтов. В таком случае "добавление 1 к указателю" означает добавление одной *единицы хранения*. Для массивов данный факт означает, что адрес увеличивается до адреса следующего *элемента*, а не просто до следующего байта (рис. 10.3). Это одна из причин того, почему нужно объявлять вид объекта, на который указывает указатель. Одного лишь адреса недостаточно, т.к. компьютер должен знать, сколько байтов требуется для хранения объекта. (Это справедливо даже для указателей на скалярные переменные; иначе операция *pt, извлекающая значение, не будет корректно работать.)

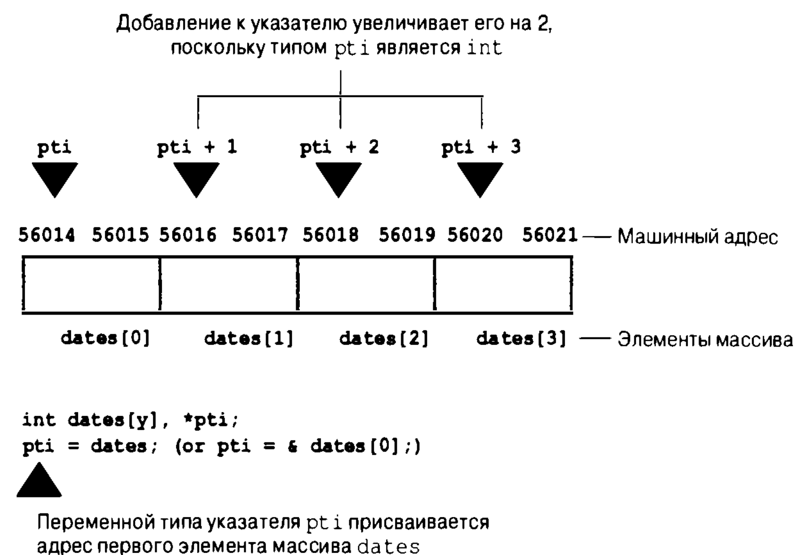


Рис. 10.3. *Массив и добавление к указателю*

Теперь мы можем более четко определить, что означают понятия "указатель на int", "указатель на float" и "указатель на любой другой объект данных".

- Значение указателя – это адрес объекта, на который он указывает. Внутреннее представление адреса зависит от оборудования. Многие компьютеры, включая IBM PC и Macintosh, *адресуемы по байтам*, т.е. байты памяти нумеруются последовательно. Адресом большого объекта, такого как переменная типа double, как правило, является адрес первого байта объекта.

- Применение операции * к указателю дает значение, хранящееся в объекте, на который ссылается указатель.
- Добавление 1 к указателю увеличивает его значение на размер в байтах типа, на который он указывает.

Мастерство языка C позволяет обеспечивать следующие равенства:

```
*dates + 2 == &date[2]    // тот же адрес
*(dates + 2) == dates[2]  // то же значение
```

Такие отношения подводят итог под тесной связью между массивами и указателями. Это значит, что вы можете использовать указатель для идентификации отдельного элемента массива и для получения его значения. По существу мы имеем две разных формы записи для одного и того же действия. В действительности, стандарт языка C описывает массивы в терминах указателей. То есть стандарт определяет ar[n] как *(ar + n). Второе выражение можно интерпретировать как "перейти к ячейке памяти ar, переместиться на n единиц и извлечь хранящееся там значение".

Кстати, не путайте *(dates+2) с *dates+2. Операция разыменования (*) имеет более высокий приоритет, чем операция +, так что второе выражение означает (*dates)+2:

```
*(dates + 2)  // значение 3-го элемента массива dates
*dates + 2    // добавление 2 к значению 1-го элемента
```

Наличие такой связи между массивами и указателями позволяет применять при написании программы любой из подходов. Например, программа в листинге 10.9 после компиляции и запуска генерирует тот же вывод, что и программа из листинга 10.1.

Листинг 10.9. Программа day_mon3.c

```
/* day_mon3.c -- использование формы записи с указателями */
#include <stdio.h>
#define MONTHS 12

int main(void)
{
   int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
   int index;

   for (index = 0; index < MONTHS; index++)
      printf("Месяц %2d имеет %d день (дней).\n", index +1,
            *(days + index));  // то же самое, что и days[index]
   return 0;
}
```

Здесь days — это адрес первого элемента массива, индекс days + index — адрес элемента days[index] и *(days + index) — значение этого элемента, в точности как days[index]. Цикл по очереди ссылается на каждый элемент массива и выводит обнаруженное содержимое.

Есть ли какое-то преимущество в написании программы подобным образом? Вообще говоря, нет — для любой формы записи компилятор генерирует один и тот же код. Основная цель кода в листинге 10.9 состояла в том, чтобы показать, что формы записи через массивы и через указатели являются эквивалентными. Этот пример демонстрирует возможность использования формы записи с указателями при работе с массивами. Обратное утверждение также верно; при работе с указателями можно применять форму записи с массивами. Это становится важным, когда имеется функция, принимающая массив в качестве аргумента.

Функции, массивы и указатели

Предположим, что необходимо написать функцию, которая оперирует на массиве. Например, пусть нужна функция, возвращающая сумму элементов массива. Представим, что marbles — это имя массива значений int. Как будет выглядеть вызов такой функции? Разумно предположить, что он должен иметь следующий вид:

```
total = sum(marbles);   // возможный вызов функции
```

А каким должен быть прототип этой функции? Вспомните, что имя массива является адресом его первого элемента, так что фактический аргумент marbles, будучи адресом значения int, должен присваиваться формальному параметру, который представляет собой указатель на тип int:

```
int sum(int * ar);      // соответствующий прототип
```

Какую информацию функция sum() получает из этого аргумента? Она получает адрес первого элемента массива и узнает, что в этой ячейке она найдет значение int. Обратите внимание, что данная информация ничего не говорит о количестве элементов в массиве. Мы поставлены перед выбором одного из двух вариантов получения этой информации функцией. Первый вариант предусматривает кодирование внутри функции фиксированного размера массива:

```
int sum(int * ar)       // соответствующее определение
{
    int i;
    int total = 0;

    for( i = 0; i < 10; i++)  // предполагается наличие 10 элементов
        total += ar[i];       // ar[i] - то же самое, что и *(ar + i)
    return total;
}
```

Здесь используется тот факт, что аналогично применению указателей с именами массивов, форму записи массивов можно использовать с указателями. Кроме того, вспомните, что операция += добавляет значение своего правого операнда к левому операнду. Следовательно, total является текущей суммой элементов массива.

Определение этой функции ограничено; она будет работать только с массивами типа int, содержащими 10 элементов. Более гибкий подход предполагает передачу во втором аргументе размера массива:

```
int sum(int * ar, int n)    // более общий подход
{
    int i;
    int total = 0;

    for( i = 0; i < n; i++)  // используются n элементов
        total += ar[i];      // ar[i] - то же самое, что и *(ar + i)
    return total;
}
```

Здесь первый параметр сообщает функции, где находится массив и какой тип данных он содержит, а второй параметр уведомляет функцию о том, сколько элементов имеется в массиве.

Есть еще один момент, который необходимо отметить касательно параметров функции. В контексте прототипа или заголовка определения функции, и *только* в этом контексте, вместо int * ar можно подставить int ar[]:

```
int sum (int ar[], int n);
```

Форма `int * ar` всегда означает, что `ar` является типом указателя на `int`. Форма `int ar[]` также означает, что `ar` – тип указателя на `int`, но *лишь* тогда, когда он применяется для объявления формальных параметров. Вторая форма напоминает читателю кода о том, что `ar` не только указывает на `int`, но указывает на значение `int`, которое представляет собой элемент массива.

НА ЗАМЕТКУ! Объявление параметров массива

Поскольку имя массива — это адрес его первого элемента, фактический аргумент в виде имени массива требует, чтобы соответствующий формальный аргумент был указателем. В этом и только в этом контексте С интерпретирует `int ar[]` как `int * ar`, т.е. `ar` является типом указателя на `int`. Поскольку в прототипах разрешено опускать имя, все четыре приведенных ниже прототипа эквивалентны:

```
int sum(int *ar, int n);
int sum(int *, int);
int sum(int ar[], int n);
int sum(int [], int);
```

В определениях функций имена опускать нельзя, поэтому следующие две формы определения эквивалентны:

```
int sum(int *ar, int n)
{
   // здесь находится код
}
int sum(int ar[], int n);
{
   // здесь находится код
}
```

Вы должны иметь возможность использовать любой из четырех показанных выше прототипов с любым из двух приведенных определений.

В листинге 10.10 показана программа, в которой применяется функция `sum()`. Чтобы отразить интересный факт, касающийся аргументов типа массива, в ней также выводится размер исходного массива и размер параметра функции, представляющего массив. (Если ваш компилятор не поддерживает спецификатор `%zd`, для вывода значений функции `sizeof` используйте спецификатор `%u` или, возможно, `%lu`.)

Листинг 10.10. Программа `sum_arr1.c`

```
// sum_sum_arr1.c -- сумма элементов массива
// используйте спецификаторы %u или %lu, если %zd не работает
#include <stdio.h>
#define SIZE 10
int sum(int ar[], int n);
int main(void)
{
   int marbles[SIZE] = {20,10,5,39,4,16,19,26,31,20};
   long answer;

   answer = sum(marbles, SIZE);
   printf("Общая сумма элементов массива marbles равна %ld.\n", answer);
   printf("Объем памяти, отведенной под массив marbles, составляет %zd байтов.\n",
         sizeof marbles);

   return 0;
}
```

```
int sum(int ar[], int n)     // насколько велик массив?
{
   int i;
   int total = 0;
   for( i = 0; i < n; i++)
   total += ar[i];
   printf("Размер ar составляет %zd байтов.\n", sizeof ar);
   return total;
}
```

Вывод в нашей системе имеет следующий вид:

```
Размер ar составляет 8 байтов.
Общая сумма элементов массива marbles равна 190.
Объем памяти, отведенной под массив marbles, составляет 40 байтов.
```

Обратите внимание, что размер массива marbles равен 40 байтов. Это имеет смысл, т.к. массив marbles содержит 10 значений типа int, каждое из которых занимает 4 байта, что в сумме составляет 40 байт. Но размер ar равен всего 8 байтов. Причина в том, что ar — это не сам массив, а указатель на первый элемент marbles. В нашей системе для хранения адресов применяются 8 байтов, поэтому размером переменной типа указателя будет 8 байтов. (В других системах может использоваться другое количество байтов.) Короче говоря, в листинге 10.10 имя marbles — это массив, ar — указатель на первый элемент массива marbles, а связь между массивами и указателями в языке С позволяет применять форму записи массива вместе с указателем ar.

Использование параметров типа указателей

Функция, работающая с массивом, должна знать, где начинать и где заканчивать свое действие. В функции sum() используется параметр типа указателя для идентификации начала массива и целочисленный параметр, отражающий количество элементов массива, которые нужно обработать. (Параметр типа указателя также описывает тип данных в массиве.) Но это не единственный способ сообщения функции того, что она должна знать. Другой способ описания массива предусматривает передачу функции двух указателей, первый из которых отмечает, где массив начинается (как и раньше), а второй — где он заканчивается. Этот подход иллюстрируется в листинге 10.11. Здесь также задействован тот факт, что параметр типа указателя является переменной, так что вместо применения индекса для сообщения о том, к какому элементу массива обращаться, в функции можно изменять значение самого указателя, заставляя его по очереди указывать на каждый элемент массива.

Листинг 10.11. Программа sum_arr2.c

```
// sum_sum_arr2.c -- сумма элементов массива
#include <stdio.h>
#define SIZE 10
int sump(int * start, int * end);
int main(void)
{
   int marbles[SIZE] = {20,10,5,39,4,16,19,26,31,20};
   long answer;

   answer = sump(marbles, marbles + SIZE);
   printf("Общая сумма элементов массива marbles равна %ld.\n", answer);
   return 0;
}
```

```
/* использование арифметики указателей */
int sump(int * start, int * end)
{
   int total = 0;
   while (start < end)
   {
      total += *start;  // добавить значение к total
      start++;          // переместить указатель на следующий элемент
   }
   return total;
}
```

Указатель `start` начинает со ссылки на первый элемент `marbles`, поэтому выражение присваивания `total +=*start` добавляет к `total` значение первого элемента (20). Затем выражение `start++` инкрементирует переменную `start`, в результате чего она указывает на следующий элемент в массиве. Поскольку `start` указывает на тип `int`, ее значение увеличивается на размер типа `int`.

Обратите внимание, что для завершения цикла суммирования в функции `sump()` используется метод, отличающийся от применяемого в `sum()`. В качестве второго аргумента функции `sum()` выступает количество элементов, и это значение используется как часть проверки конца цикла:

```
for( i = 0; i < n; i++)
```

Однако в `sump()` для проверки окончания цикла применяется второй указатель:

```
while (start < end)
```

Поскольку производится проверка на предмет неравенства, последним обработанным элементом массива будет элемент, находящийся непосредственно перед элементом, на который указывает end. Это означает, что end в действительности указывает на ячейку, расположенную после финального элемента массива. В языке С гарантируется, что при выделении пространства памяти под массив указатель на первую ячейку после конца массива будет допустимым. Благодаря этому, конструкция подобного рода является допустимой, т.к. последним значением, которое `start` получает в цикле, будет end. Обратите внимание, что использование такого указателя на место “за пределами конца массива”, делает вызов функции лаконичным:

```
answer = sump(marbles, marbles + SIZE);
```

Из-за того, что индексация начинается с 0, `marbles + SIZE` указывает на элемент, следующий за концом массива. Если бы end указывал на последний элемент, а не на следующий за концом массива, пришлось бы применять такой код:

```
answer = sump(marbles, marbles + SIZE - 1);
```

Подобный код не только внешне менее элегантен, его еще и труднее запомнить, поэтому с большей вероятностью можно допустить ошибку. Между прочим, хотя язык С гарантирует допустимость указателя `marbles + SIZE`, нет никаких гарантий в отношении `marbles[SIZE]`, т.е. значения, хранящегося в этой ячейке, поэтому программа не должна пытаться получить доступ к ней.

Тело цикла можно ужать до одной строки:

```
total += *start++;
```

Унарные операции * и ++ имеют один и тот же приоритет, но ассоциацию справа налево. Это означает, что операция ++ применяется к `start`, а не к `*start`.

Другими словами, инкрементируется указатель, а не значение, на которое он указывает. Использование постфиксной формы (start++ вместо ++start) приведет к тому, что указатель не инкрементируется до тех пор, пока указываемое им значение не будет добавлено к total. Если бы в программе применялось выражение *++start, то сначала бы инкрементировался указатель, а затем использовалось бы значения, на которое он указывает. Однако если бы в программе было задействовано выражение (*start)++, то сначала использовалось бы значение start и затем инкрементировалось бы значение, а не указатель. Тогда указатель остался бы нацеленным на тот же самый элемент, но элемент содержал бы новое число. Хотя обычно применяется запись *start++, форма *(start++) более понятна. В листинге 10.12 демонстрируются особенности приоритетов.

Листинг 10.12. Программа order.c

```
/* order.c -- приоритеты в операциях с указателями */
#include <stdio.h>
int data[2] = {100, 200};
int moredata[2] = {300, 400};
int main(void)
{
    int * p1, * p2, * p3;

    p1 = p2 = data;
    p3 = moredata;
    printf(" *p1 = %d, *p2 = %d, *p3 = %d\n",
            *p1    , *p2    , *p3);
    printf("*p1++ = %d, *++p2 = %d, (*p3)++ = %d\n",
            *p1++   , *++p2    , (*p3)++);
    printf(" *p1 = %d, *p2 = %d, *p3 = %d\n",
            *p1    , *p2    , *p3);

    return 0;
}
```

Вот вывод, полученный в результате запуска программы:

```
  *p1 = 100,   *p2 = 100,     *p3 = 300
*p1++ = 100, *++p2 = 200, (*p3)++ = 300
  *p1 = 200,   *p2 = 200,     *p3 = 301
```

Единственной операцией, которая изменяет значение массива, является (*p3)++. Другие две операции приводят к тому, что p1 и p2 начинают указывать на следующий элемент массива.

Комментарии: указатели и массивы

Как вы уже видели, функции, которые обрабатывают массивы, в действительности используют указатели в качестве аргументов, но при написании функций обработки массивов вы должны сделать выбор между формой записи в виде массива и формой записи посредством указателей. Применение формы записи для массивов, как в листинге 10.10, делает более очевидным тот факт, что функция работает с массивами. Кроме того, такая форма записи более привычна для программистов, перешедших с других языков, таких как FORTRAN, Pascal, Modula-2 или BASIC. Другие программисты могут быть больше приучены к работе с указателями и посчитают более естественной форму записи с использованием указателей вроде показанной в листинге 10.11.

Что касается языка С, то два выражения ar[i] и *(ar+i) по смыслу эквивалентны. Оба работают, если ar является именем массива, и оба работают, если ar — это переменная типа указателя. Тем не менее, выражение наподобие ar++ работает только в тех случаях, когда ar представляет собой переменную типа указателя.

Запись с применением указателей, особенно когда она сопровождается операцией инкремента, ближе к машинному языку, и некоторые компиляторы обеспечивают в таком случае более эффективный код. Однако многие программисты придерживаются мнения о том, что их основная задача заключается в обеспечении корректности и ясности кода, а его оптимизация должна быть оставлена компилятору.

Операции с указателями

Что же разрешено делать с указателями? Язык С предлагает множество базовых операций, которые можно выполнять над указателями, и в следующей программе демонстрируются восемь из имеющихся возможностей. Чтобы показать результаты каждой операции, программа выводит значение указателя (адрес, на который он указывает), значение, хранящееся по указанному адресу, и адрес самого указателя. (Если ваш компилятор не поддерживает спецификатор %p, попробуйте воспользоваться для вывода адресов спецификатором %u или, возможно, %lu. Если компилятор не поддерживает спецификатор %td, предназначенный для вывода разности адресов, попробуйте применить %d или, возможно, %ld.)

В листинге 10.13 представлены восемь базовых операций, которые можно выполнять над переменными типа указателя. В дополнение к этим операциям можно использовать операции отношений для сравнения указателей.

Листинг 10.13. Программа ptr_ops.c

```
// ptr_ops.c -- операции над указателями
#include <stdio.h>
int main(void)
{
   int urn[5] = {100,200,300,400,500};
   int * ptr1, * ptr2, *ptr3;

   ptr1 = urn;              // присваивание указателю адреса
   ptr2 = &urn[2];          // то же самое
                            // разыменование указателя и получение
                            // адреса указателя
   printf("значение указателя, разыменованный указатель, адрес указателя:\n");
   printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n",
          ptr1, *ptr1, &ptr1);
   // сложение указателей
   ptr3 = ptr1 + 4;
   printf("\nсложение значения int с указателем:\n");
   printf("ptr1 + 4 = %p, *(ptr4 + 3) = %d\n",
          ptr1 + 4, *(ptr1 + 3));
   ptr1++;                  // инкрементирование указателя
   printf("\nзначения после выполнения операции ptr1++:\n");
   printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n",
          ptr1, *ptr1, &ptr1);
   ptr2--;                  // декрементирование указателя
   printf("\nзначения после выполнения операции --ptr2:\n");
   printf("ptr2 = %p, *ptr2 = %d, &ptr2 = %p\n",
          ptr2, *ptr2, &ptr2);
   --ptr1;                  // восстановление исходного значения
   ++ptr2;                  // восстановление исходного значения
```

```
    printf("\nвосстановление исходных значений указателей:\n");
    printf("ptr1 = %p, ptr2 = %p\n", ptr1, ptr2);

    // вычитание одного указателя из другого
    printf("\nвычитание одного указателя из другого:\n");
    printf("ptr2 = %p, ptr1 = %p, ptr2 - ptr1 = %td\n",
          ptr2, ptr1, ptr2 - ptr1);

    // вычитание целого значения из указателя
    printf("\nвычитание из указателя значения типа int:\n");
    printf("ptr3 = %p, ptr3 - 2 = %p\n", ptr3, ptr3 - 2);

    return 0;
}
```

Ниже показаны результаты выполнения этой программы в одной из систем:

```
значение указателя, разыменованный указатель, адрес указателя:
ptr1 = 0x7fff5fbff8d0, *ptr1 =100, &ptr1 = 0x7fff5fbff8c8

сложение значения int с указателем:
ptr1 + 4 = 0x7fff5fbff8e0, *(ptr4 + 3) = 400

значения после выполнения операции ptr1++:
ptr1 = 0x7fff5fbff8d4, *ptr1 =200, &ptr1 = 0x7fff5fbff8c8

значения после выполнения операции --ptr2:
ptr2 = 0x7fff5fbff8d4, *ptr2 = 200, &ptr2 = 0x7fff5fbff8c0

восстановление исходных значений указателей:
ptr1 = 0x7fff5fbff8d0, ptr2 = 0x7fff5fbff8d8

вычитание одного указателя из другого:
ptr2 = 0x7fff5fbff8d8, ptr1 = 0x7fff5fbff8d0, ptr2 - ptr1 = 2

вычитание из указателя значения типа int:
ptr3 = 0x7fff5fbff8e0, ptr3 - 2 = 0x7fff5fbff8d8
```

Ниже описаны базовые операции, которые можно выполнять с переменными типа указателей.

- **Присваивание.** Указателю можно присвоить адрес. Присваиваемым значением может быть, например, имя массива, переменная, которой предшествует операция взятия адреса (&), или другой указатель. В листинге 10.13 переменной ptr1 присваивается адрес начала массива urn. Этим адресом оказался номер ячейки памяти 0x7fff5fbff8d0. Переменная ptr2 получает адрес третьего (последнего) элемента, urn[2]. Обратите внимание, что адрес должен быть совместим с типом указателя. Другими словами, вы не можете присваивать адрес значения double указателю на int, во всяком случае, не делая неосмотрительное приведение типа. Это правило требуют стандарты C99/C11.

- **Нахождение значения (разыменование).** Операция * дает значение, хранящееся в ячейке, на которую указывает указатель. Таким образом, первоначально *ptr1 равно 100, т.е. значению, хранящемуся в ячейке 0x7fff5fbff8d0.

- **Взятие адреса указателя.** Подобно всем переменным, переменная типа указателя имеет адрес и значение. Операция & сообщает, где хранится сам указатель. В рассмотренном примере ptr1 хранится в ячейке 0x7fff5fbff8c8. Содержимым этой ячейки памяти является 0x7fff5fbff8d0, т.е. адрес массива urn. В итоге &pt1 – указатель на pt1, что, в свою очередь, представляет собой указатель на urn[0].

- **Добавление целого числа к указателю.** С помощью операции + можно добавить целое число к указателю или указатель к целому числу. В любом случае целое число умножается на количество байтов в типе данных, на который указывает указатель, и результат добавляется к исходному адресу. Это делает выражение `ptr1 + 4` эквивалентным `&urn[4]`. Результат сложения не определен, если он находится за пределами массива, на который указывает исходный указатель; исключением будет адрес, следующий за последним элементом массива, который считается допустимым.

- **Инкрементирование указателя.** Инкрементирование указателя на элемент массива приводит к его перемещению на следующий элемент массива. Таким образом, операция `ptr1++` увеличивает числовое значение `ptr1` на 4 (4 байта для типа `int` в нашей системе) и указатель `ptr1` будет ссылаться на `urn[1]` (на рис. 10.4 приведена иллюстрация с применением упрощенных адресов). Теперь `ptr1` имеет значение `0x7fff5fbff8d4` (адрес следующего элемента в массиве), а `*ptr1` — значение `200` (значение `urn[1]`). Обратите внимание, что адресом самого `ptr1` остается `0x7fff5fbff8c8`. В конце концов, переменная не перемещается в памяти лишь потому, что изменилось ее значение!

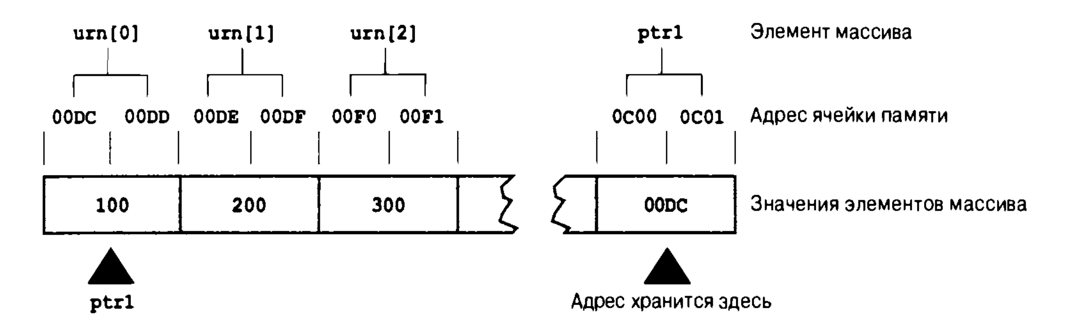


Рис. 10.4. Инкрементирование указателя на `int`

- **Вычитание целого числа из указателя.** Посредством операции - можно вычитать целое число из указателя; указатель должен быть первым операндом, а целое число — вторым. Целое число умножается на количество байтов в типе, на который указывает указатель, и результат вычитается из исходного адреса. Это делает `ptr3 - 2` эквивалентным `&urn[2]`, т.к. `ptr3` указывает на `&urn[4]`. Результат вычитания не определен, если он находится за пределами массива, на который указывает исходный указатель; исключением будет адрес, следующий за последним элементом массива, который считается допустимым.

- **Декрементирование указателя.** Разумеется, указатель можно также декрементировать. В приведенном примере декрементирование `ptr2` приводит к тому, что он указывает на второй элемент массива вместо третьего. Обратите внимание, что можно использовать как префиксную, так и постфиксную форму операций инкремента и декремента. Также следует отметить, что перед восстановлением исходных значений `ptr1` и `ptr2` указывали на один и тот же элемент, `urn[1]`.

- **Разность.** Вы можете находить разность между двумя указателями. Обычно это делается для двух указателей на элементы, находящиеся в одном массиве, чтобы определить, насколько далеко они отстоят друг от друга. Результат представлен в тех же единицах, что и размер типа. Например, в выводе программы из листинга 10.13 выражение `ptr2 - ptr1` имеет значение `2`, т.е. эти указатели ссылаются на объекты, которые отделены друг от друга двумя значениями `int`, а не двумя байтами. Вычитание является гарантированно допустимой операцией при условии, что оба указателя ссылаются на значения внутри одного и того же массива (или, возможно, на позицию за последним элементом массива). Применение этой операции к указателям в двух разных массивах может дать какое-то значение или привести к ошибке во время выполнения.
- **Сравнение.** Для сравнения значений двух указателей можно использовать операции отношений при условии, что указатели имеют один и тот же тип.

Обратите внимание на существование двух форм вычитания. Можно вычитать один указатель из другого и получать целое число, а также можно вычитать целое число из указателя и получать указатель.

При выполнении инкрементирования и декрементирования указателя необходимо соблюдать определенные меры предосторожности. Компьютер не отслеживает, продолжает ли указатель ссылаться на элемент в массиве. Язык С гарантирует допустимость указателя, если он ссылается на любой элемент заданного массива или на позицию, следующую за последним элементом массива. Но результат инкрементирования или декрементирования указателя, который выходит за эти пределы, не определен. Кроме того, можно разыменовать указатель на любой элемент массива. Однако, несмотря на допустимость указателя, ссылающегося на позицию после конца массива, возможность его разыменования не гарантируется.

Разыменование неинициализированного указателя

Говоря об осторожности, существует одно правило, о котором вы не должны забывать: никогда не разыменовывайте неинициализированный указатель! Например, взгляните на следующий код:

```
int * pt;  // неинициализированный указатель
*pt = 5;   // катастрофическая ошибка
```

Почему здесь все настолько плохо? Вторая строка означает сохранение значения `5` в ячейке, на которую указывает `pt`. Но `pt`, будучи неинициализированным, имеет случайное значение, поэтому неизвестно, куда будет помещено `5`. Это может не причинить вреда, перезаписать данные или код либо вызвать аварийное завершение программы. Вспомните, что создание указателя приводит к выделению памяти только под сам указатель; для хранения данных память не выделяется. Таким образом, перед использованием указателю должен быть присвоен адрес ячейки памяти, которая уже была выделена. Например, указателю можно присвоить адрес существующей переменной. (Именно это происходит во время применения функции с параметром типа указателя.) Либо же можно воспользоваться функцией `malloc()` для предварительного выделения памяти, как обсуждается в главе 12. В любом случае, во избежание проблем, никогда не разыменовывайте неинициализированный указатель!

```
double * pd;  // неинициализированный указатель
*pd = 2.4;    // НЕ ПОСТУПАЙТЕ ТАК!
```

Предположим, что есть такой код:

```
int urn[3];
int * ptr1, * ptr2;
```

Ниже приведены примеры допустимых и недопустимых операторов:

Допустимо	Недопустимо
`ptr1++;`	`urn++;`
`ptr2 = ptr1 + 2;`	`ptr2 = ptr2 + ptr1;`
`ptr2 = urn + 1;`	`ptr2 = urn * ptr1;`

Допустимые операции открывают множество возможностей. Программисты на C создают массивы указателей, указатели на функции, массивы указателей на указатели, массивы указателей на функции и т.п. Однако мы будем придерживаться базовых случаев применения указателей, которые рассматривались выше. Первый базовый случай использования указателей связан с передачей информации в и из функций. Вы уже знаете, что для воздействия внутри функции на переменные из вызывающей функции должны применяться указатели. Второй случай использования касается функций, предназначенных для манипулирования массивами. Давайте взглянем на еще один пример, в котором применяются функции и массивы.

Защита содержимого массива

При написании функции, которая обрабатывает фундаментальный тип вроде int, у вас есть выбор между передачей данных int по значению и передачей указателя на тип int. Обычно числовые данные передаются по значению, если только программа не нуждается в изменении этого значения — в таком случае передается указатель. Массивы не предоставляют подобного выбора; вы *обязаны* передавать указатель. Все дело в эффективности. Если бы массив передавался по значению, пришлось бы выделять в памяти пространство, достаточное для сохранения копии исходного массива, и затем копировать все данные из исходного массива в новый. Намного быстрее передать адрес массива и заставить функцию работать с исходными данными.

Такой подход может привести к проблемам. Причиной обычной передачи данных по значению является обеспечение целостности данных. Если функция работает с копией исходных данных, она не сможет случайно исказить эти данные. Но поскольку функции обработки массивов работают с исходными данными, они *могут* модифицировать массив. Временами это желательно. Например, ниже приведена функция, которая добавляет одно и то же значение к каждому элементу массива:

```
void add_to(double ar[], int n, double val)
{
   int i;
   for( i = 0; i < n; i++)
      ar[i] += val;
}
```

Следовательно, вызов функции

```
add_to(prices, 100, 2.50);
```

приводит к тому, что каждый элемент массива prices заменяется значением, превосходящим прежнее значение на 2.5; эта функция изменяет содержимое массива. Функция может делать это потому, что за счет работы с указателями она имеет дело с исходными данными.

Тем не менее, другие функции не предназначены для модификации данных. Например, показанная далее функция подсчитывает сумму содержимого массива; она

не должна изменять массив. Но поскольку ar в действительности представляет собой указатель, ошибка в коде может вызвать повреждение исходных данных. К примеру, здесь выражение ar[i]++ в результате приводит к увеличению на 1 значения каждого элемента:

```
int sum(int ar[], int n)    // ошибочный код
{
   int i;
   int total = 0;

   for( i = 0; i < n; i++)
      total += ar[i]++;   // ошибочное инкрементирование каждого элемента
   return total;
}
```

Использование const с формальными параметрами

В языке K&R C единственный способ избежать ошибки такого рода — быть внимательным. С выходом ANSI C появилась альтернатива. Если функция не задумывалась как изменяющая содержимое массива, применяйте ключевое слово const при объявлении формального параметра в прототипе и в определении функции. Например, прототип и определение функций sum() должны иметь следующий вид:

```
int sum(const int ar[], int n);  /* прототип */

int sum(const int ar[], int n)   /* определение */
{
   int i;
   int total = 0;

   for( i = 0; i < n; i++)
      total += ar[i];
   return total;
}
```

Это сообщает компилятору о том, что функция должна трактовать массив, указанный посредством ar, как содержащий константные данные. Если затем вы случайно воспользуетесь выражением, подобным ar[i]++, компилятор сумеет обнаружить это и сгенерировать сообщение об ошибке, уведомляющее о том, что функция пытается изменить константные данные.

Важно понимать, что такое применение ключевого слова const вовсе не требует, чтобы исходный массив *был* константным; оно лишь говорит о том, что функция должна трактовать массив так, как *если бы* он был константным. Использование const в подобной манере предоставляет защиту для массивов, которую обеспечивает фундаментальным типам передача по значению; оно предотвращает модификацию внутри функции данных из вызывающей функции. В общем случае при написании функции, предназначенной для изменения массива, не указывайте const при объявлении параметра типа массива. Если же вы пишете функцию, не предназначенную для модификации массива, применяйте ключевое слово const при объявлении параметра типа массива.

В программе, показанной в листинге 10.14, одна функция отображает массив, а другая умножает каждый элемент массива на заданное значение. Поскольку первая функция не должна изменять массив, в ней используется const. Из-за того, что вторая функция намерена модифицировать массив, ключевое слово const в ней отсутствует.

Листинг 10.14. Программа arf.c

```
/* arf.c -- функции, манипулирующие массивами */
#include <stdio.h>
#define SIZE 5
void show_array(const double ar[], int n);
void mult_array(double ar[], int n, double mult);
int main(void)
{
   double dip[SIZE] = {20.0, 17.66, 8.2, 15.3, 22.22};

   printf("Исходный массив dip:\n");
   show_array(dip, SIZE);
   mult_array(dip, SIZE, 2.5);
   printf("Массив dip после вызова функции mult_array():\n");
   show_array(dip, SIZE);

   return 0;
}

/* выводит содержимое массива */
void show_array(const double ar[], int n)
{
   int i;

   for (i = 0; i < n; i++)
      printf("%8.3f ", ar[i]);
   putchar('\n');
}

/* умножает каждый элемент массива на один и тот же множитель */
void mult_array(double ar[], int n, double mult)
{
   int i;
   for (i = 0; i < n; i++)
      ar[i] *= mult;
}
```

Ниже приведен вывод:

```
Исходный массив dip:
  20.000  17.660   8.200  15.300  22.220
Массив dip после вызова функции mult_array():
  50.000  44.150  20.500  38.250  55.550
```

Обратите внимание, что типом обеих функций является void. Функция mult_array() предоставляет новые значения массиву dip, но не за счет применения механизма return.

Дополнительные сведения о ключевом слове const

Вы уже знаете, что ключевое слово const можно использовать для создания символических констант:

```
const double PI = 3.14159;
```

То же самое можно было бы сделать с помощью директивы #define, но ключевое слово const дополнительно позволяет создавать константные массивы, константные указатели и указатели на константы.

В листинге 10.4 продемонстрировано применение ключевого слова const для защиты массива от модификации:

```
#define MONTHS 12
...
const int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Если впоследствии попытаться в коде изменить массив, на этапе компиляции будет сгенерировано сообщение об ошибке:

```
days[9] = 44;              /* ошибка на этапе компиляции */
```

Указатели на константы не могут использоваться для изменения значений. Взгляните на следующий код:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double * pd = rates;     // pd указывает на начало массива
```

Вторая строка кода объявляет, что значение типа double, на которое указывает pd, является const. Это означает, что pd нельзя применять для изменения значений, на которые он указывает:

```
*pd = 29.89;               // не разрешено
pd[2] = 222.22;            // не разрешено
rates[0] = 99.99;          // разрешено, т.к. rates не является const
```

Независимо от того, какая форма записи используется — с указателем или с массивом, pd не разрешено применять для изменения данных, на которые он указывает. Тем не менее, поскольку массив rates не был объявлен как константа, вы можете по-прежнему использовать rates для изменения значений. Кроме того, обратите внимание, что можно сделать так, чтобы pd указывал на что-нибудь другое:

```
pd++;                     /* теперь pd указывает на rates[1] -- разрешено */
```

Указатель на константу обычно передается в виде параметра функции для сообщения о том, что функция не будет его применять в целях изменения данных. Например, функция show_array() из листинга 10.14 могла бы иметь такой прототип:

```
void show_array(const double *ar, int n);
```

Существует несколько правил, которые вы должны соблюдать, присваивая указатели и используя ключевое слово const. Прежде всего, указателю на константу допускается присваивание адреса либо константных, либо неконстантных данных:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double locked[4] = {0.08, 0.075, 0.0725, 0.07};
const double * pc = rates; // допустимо
pc = locked;               // допустимо
pc = &rates[3];            // допустимо
```

Тем не менее, обычным указателям могут быть присвоены только адреса неконстантных данных:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double locked[4] = {0.08, 0.075, 0.0725, 0.07};
double * pnc = rates;      // допустимо
pnc = locked;              // не допустимо
pnc = &rates[3];           // допустимо
```

Это обоснованное правило. Иначе указатель можно было бы применять для изменения данных, которые должны быть константными.

Практическим следствием этих правил является то, что функция наподобие show_array() может принимать в качестве фактических аргументов имена обычных массивов *и* константных массивов, поскольку каждый из них может быть присвоен указателю на константу:

```
show_array(rates, 5);          // допустимо
show_array(locked, 4);         // допустимо
```

Таким образом, использование ключевого слова const в определении параметра функции не только защищает данные, но также позволяет функции работать с массивами, которые были объявлены как const.

Однако функции вроде mult_array() не должно передаваться имя константного массива в виде аргумента:

```
mult_array(rates, 5, 1.2);     // допустимо
mult_array(locked, 4, 1.2);    // не допустимо
```

В стандарте С говорится о том, что попытка модификации данных const, таких как locked, с применением отличного от const идентификатора, например, формального аргумента ar функции mult_array(), приводит к неопределенному поведению.

Существуют и другие варианты использования const. К примеру, вы можете объявить и инициализировать указатель таким образом, чтобы его нельзя было заставить указывать на что-нибудь другое. Хитрость в том, где размещено ключевое слово const:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
double * const pc = rates;     // pc указывает на начало массива
pc = &rates[2];                // не разрешено указывать на что-нибудь другое
*pc = 92.99;                   // все в порядке -- изменяется rates[0]
```

Такой указатель можно по-прежнему применять для изменения значений, но он может указывать только на ячейку, которая была присвоена первоначально.

Наконец, const можно использовать дважды, чтобы создать указатель, который не допускает изменения ни адреса, куда он указывает, ни указываемого с помощью него значения:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double * const pc = rates;
pc = &rates[2];                // не разрешено
pc = &rates[92.99];            // не разрешено
```

Указатели и многомерные массивы

Каким образом указатели связаны с многомерными массивами? И для чего это необходимо знать? Функции, которые работают с многомерными массивами, делают это с помощью указателей, поэтому прежде чем переходить к исследованию таких функций, нужно продолжить изучение указателей. Чтобы получить ответ на первый вопрос, рассмотрим несколько примеров. Для простоты ограничимся массивом небольшого размера. Предположим, что имеется следующее объявление:

```
int zippo[4][2];  /* массив из массивов типа int */
```

Тогда zippo, будучи именем массива, представляет собой адрес первого элемента в этом массиве. В данном случае первый элемент zippo сам является массивом из двух значений int, так что zippo — это адрес массива, содержащего два значения int.

Давайте проанализируем это дополнительно в терминах свойств указателей.

- Так как zippo – адрес первого элемента массива, zippo имеет то же значение, что и &zippo[0]. Вдобавок zippo[0] сам по себе является массивом из двух целых чисел, следовательно, zippo[0] имеет то же значение, что и &zippo[0][0], т.е. адрес его первого элемента – значения int. Короче говоря, zippo[0] – это адрес объекта с размером значения int, а zippo – адрес объекта с размером двух значений int. Поскольку и целое число, и массив из двух целых чисел начинаются в одной и той же позиции, числовые значения zippo и zippo[0] одинаковы.
- Добавление 1 к указателю или адресу дает значение, которое больше исходного на размер указываемого объекта. В этом отношении zippo и zippo[0] отличаются друг от друга, потому что zippo ссылается на объект с размером в два значения int, а zippo[0] – на объект с размером в одно значение int. Таким образом, zippo + 1 имеет значение, не совпадающее с zippo[0] + 1.
- Разыменование указателя или адреса (применение операции * или операции [] с индексом) дает значение, представленное объектом, на который производится ссылка. Поскольку zippo[0] – адрес его первого элемента (zippo[0][0]), то *(zippo[0]) представляет значение, хранящееся в zippo[0][0], т.е. значение int. Аналогично, *zippo представляет значение своего первого элемента (zippo[0]), но zippo[0] сам по себе – адрес значения int. Это адрес &zippo[0][0], так что *zippo является &zippo[0][0]. Применение операции разыменования к обоим выражениям предполагает, что **zippo равно *&zippo[0][0], что сокращается до zippo[0][0], т.е. значения типа int. Короче говоря, zippo – это адрес адреса, и для получения обычного значения потребуется двукратное разыменование. Адрес адреса или указатель на указатель представляют собой примеры *двойной косвенности.*

Очевидно, что увеличение количества измерений массива повышает сложность представления с помощью указателей. На этом этапе большинство людей, изучающих С, начинают понимать причины, по которым указатели считаются одним из наиболее трудных аспектов языка. Возможно, вам потребуется еще раз почитать о свойствах указателей, которые описаны выше, после чего обратиться к листингу 10.15, где отображаются значения некоторых адресов и содержимое массивов.

Листинг 10.15. Программа zippo1.c

```
/* zippo1.c -- информация о массиве zippo */
#include <stdio.h>
int main(void)
{
   int zippo[4][2] = { {2,4}, {6,8}, {1,3}, {5, 7} };
   printf(" zippo = %p, zippo + 1 = %p\n",
          zippo,    zippo + 1);
   printf("zippo[0] = %p, zippo[0] + 1 = %p\n",
         zippo[0],    zippo[0] + 1);
   printf(" *zippo = %p, *zippo + 1 = %p\n", *zippo,    *zippo + 1);
   printf("zippo[0][0] = %d\n", zippo[0][0]);
   printf("  *zippo[0] = %d\n", *zippo[0]);
   printf("    **zippo = %d\n", **zippo);
   printf("     zippo[2][1] = %d\n", zippo[2][1]);
   printf("*(*(zippo+2) + 1) = %d\n", *(*(zippo+2) + 1));
   return 0;
}
```

Ниже показан вывод, полученный в одной из систем:

```
   zippo = 0x0064fd38,    zippo + 1 = 0x0064fd40
zippo[0] = 0x0064fd38, zippo[0] + 1 = 0x0064fd3c
  *zippo = 0x0064fd38,   *zippo + 1 = 0x0064fd3c
zippo[0][0] = 2
  *zippo[0] = 2
    **zippo = 2
      zippo[1][2] = 3
*(*(zippo+1) + 2) = 3
```

В других системах могут отображаться другие значения адресов и в отличающихся форматах, но взаимосвязи будут такими же, как описано в настоящем разделе. Вывод показывает, что адреса двумерного массива zippo и одномерного массива zippo[0] совпадают. Каждый из них является адресом первого элемента соответствующего массива, и в числовом эквиваленте имеет то же значение, что и &zippo[0][0].

Однако имеется и различие. В нашей системе тип int занимает 4 байта. Как обсуждалось ранее, zippo[0] указывает на 4-байтовый объект данных. Добавление 1 к zippo[0] должно дать значение, превышающее исходное на 4, что и было получено. (В шестнадцатеричной записи 38 + 4 равно 3c.) Имя zippo — это адрес массива из двух значений int, поэтому он идентифицирует 8-байтовый объект данных. Таким образом, добавление 1 к zippo должно привести к адресу, который на 8 байтов больше исходного, что и происходит на самом деле. (В шестнадцатеричной записи 40 на 8 больше, чем 38.)

Программа демонстрирует, что zippo[0] и *zippo идентичны, как и должно быть. Затем она показывает, что для получения хранящегося в массиве значения имя двумерного массива должно быть разыменовано дважды. Это может быть сделано за счет двукратного применения операции разыменования (*) или операции квадратных скобок ([]). (Этого также можно достичь с использованием одной операции * и одного набора квадратных скобок, но давайте не будем отвлекаться на исследования всех возможных вариантов.)

В частности, обратите внимание, что эквивалент zippo[2][1] в форме записи с указателями выглядит как *(*(zippo+2) + 1). Вероятно, хотя бы раз в жизни вам приходилось прикладывать усилия, чтобы разобрать такое выражение. Давайте будем анализировать это выражение пошагово:

```
zippo            ←адрес первого элемента длиной в два значения int
zippo+2          ←адрес третьего элемента длиной в два значения int
*(zippo+2)       ←третий элемент, представляющий собой массив из двух int,
                  следовательно, это адрес его первого элемента, т.е. значения int
*(zippo+2) + 1   ←адрес второго элемента в массиве из двух int, также значение int
*(*(zippo+2) + 1)←значение второго int в третьей строке (zippo[2][1])
```

Смысл этой причудливой формы с указателями заключается вовсе не в том, что ее можно применять вместо более простой записи zippo[2][1]. Смысл в том, что при наличии указателя на двумерный массив и необходимости извлечь значение можно использовать более простую форму записи в виде массива, а не форму с указателями.

На рис. 10.5 показано еще одно представление отношений между адресами массива, содержимым массива и указателями.

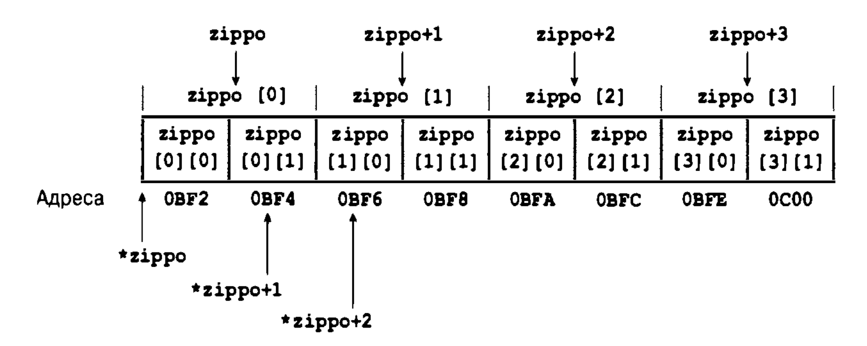


Рис. 10.5. Массив из массивов

Указатели на многомерные массивы

Как бы вы объявили переменную pz типа указателя, которая может указывать на двумерный массив, такой как zippo? Указатель подобного рода мог бы применяться, например, при написании функции, которая имеет дело с массивами вроде zippo. Достаточно ли будет типа указателя на int? Нет. Такой тип совместим с zippo[0], который указывает на одиночное значение int. Но zippo — это адрес его первого элемента, который сам является массивом из двух значений int. Отсюда следует, что pz должен указывать на массив с двумя элементами int, а не на одиночное значение int. Вот как можно поступить:

```
int (* pz)[2];  // pz указывает на массив из 2 значений int
```

Приведенный оператор определяет, что pz представляет собой указатель на массив из двух значений типа int. Для чего здесь нужны круглые скобки? Дело в том, что скобки [] имеют более высокий приоритет, чем *. Это значит, что в объявлении вида

```
int * pax[2];  // pax - массив из двух указателей на int
```

сначала используются квадратные скобки, делая pax массивом с какими-то двумя элементами. Затем применяется операция *, превращая pax в массив из двух указателей. Наконец, использование int делает pax массивом из двух указателей на int. Приведенное объявление создает *два* указателя на одиночные значения int, но в первоначальной версии круглые скобки обеспечивают применение операции * первой, создавая *один* указатель на массив из двух значений int. В листинге 10.16 показано, что такой указатель можно использовать подобно исходному массиву.

Листинг 10.16. Программа zippo2.c

```
/* zippo2.c -- получение информации о массиве zippo с помощью переменной типа
указателя */
#include <stdio.h>
int main(void)
{
   int zippo[4][2] = { {2,4}, {6,8}, {1,3}, {5, 7} };
   int (*pz)[2];
   pz = zippo;

   printf(" pz = %p, pz + 1 = %p\n",
          pz,     pz + 1);
   printf("pz[0] = %p, pz[0] + 1 = %p\n",
         pz[0],     pz[0] + 1);
```

```
    printf(" *pz = %p, *pz + 1 = %p\n",
           *pz,     *pz + 1);
    printf("pz[0][0] = %d\n", pz[0][0]);
    printf("  *pz[0] = %d\n", *pz[0]);
    printf("    **pz = %d\n", **pz);
    printf("     pz[2][1] = %d\n", pz[2][1]);
    printf("*(*(pz+2) + 1) = %d\n", *(*(pz+2) + 1));

    return 0;
}
```

Вот новый вывод:

```
pz = 0x0064fd38,     pz + 1 = 0x0064fd40
pz[0] = 0x0064fd38, pz[0] + 1 = 0x0064fd3c
  *pz = 0x0064fd38,   *pz + 1 = 0x0064fd3c
pz[0][0] = 2
  *pz[0] = 2
    **pz = 2
      pz[2][1] = 3
*(*(pz+2) + 1) = 3
```

И снова в своей системе вы можете получить другие адреса, но взаимосвязи останутся такими же. Как и было обещано, форму записи наподобие pz[2][1] можно применять, даже если pz является указателем, а не именем массива. Говоря в общем, вы можете представлять отдельные элементы, используя форму записи с участием массива или указателей либо с именем массива, либо с указателем:

```
zippo[m][n] == *(*(zippo + m) + n)
pz[m][n] == *(*(pz + m) + n)
```

Совместимость указателей

Правила присваивания одного указателя другому строже таких правил для числовых типов. Например, вы можете присвоить значение int переменной double, не используя преобразование типа, но нельзя сделать то же самое для указателей на эти два типа:

```
int n = 5;
double x;
int * p1 = &n;
double * pd = &x;
x = n;              // неявное преобразование типа
pd = p1;            // ошибка на этапе компиляции
```

Такие ограничения распространяются и на более сложные типы. Предположим, что есть следующие объявления:

```
int * pt;
int (*pa)[3];
int ar1[2][3];
int ar2[3][2];
int **p2;           // указатель на указатель
```

Взгляните на показанный далее код:

```
pt = &ar1[0][0];// оба - указатели на int
pt = &ar1[0];     // оба - указатели на int
pt = ar1;         // недопустимо
pa = ar1;         // оба - указатели на int[3]
```

```
pa = ar2;        // недопустимо
pt = &pt;        // оба - указатели на int
*p2 = ar2[0];    // оба - указатели на int
p2 = ar2;        // недопустимо
```

Обратите внимание, что во всех недопустимых случаях присваивания вовлечены два указателя, которые не указывают на один и тот же тип. Например, pt указывает на одиночное значение int, но ar1 — на массив из трех значений int. Аналогично, pa указывает на массив из двух значений int, следовательно, он совместим с ar1, но не с ar2, который указывает на массив из двух значений int.

Два последних примера немного запутаны. Переменная p2 представляет собой указатель на указатель на тип int, в то время как ar2 — это указатель на массив из двух значений int (или, выражаясь короче, указатель на массив int[2]). Таким образом, p2 и ar2 — разные типы, и вы не можете присвоить ar2 указателю p2. Но *p2 имеет тип указателя на int, что обеспечивает его совместимость с ar2[0]. Вспомните, что ar2[0] является указателем на свой первый элемент, ar2[0][0], что делает ar2[0] также и типом указателя на int.

В целом, многократные операции разыменования сложны. Например, рассмотрим следующий фрагмент кода:

```
int x = 20;
const int y = 23;
int * p1 = &x;
const int * p2 = &y;
const int ** pp2;
p1 = p2;    // небезопасно -- присваивание константного значения неконстантному
p2 = p1;    // допустимо -- присваивание константного значения константному
pp2 = &p1;  // небезопасно -- присваивание вложенных типов указателей
```

Как вы видели ранее, присваивание указателя const указателю, отличному от const, не является безопасным, т.к. новый указатель мог бы применяться для изменения данных типа const. Хотя код и скомпилируется, возможно, с выдачей предупреждения, результат его выполнения не определен. Но присваивание указателя не const указателю const допустимо при условии, что вы имеете дело только с одним уровнем косвенности:

```
p2 = p1;   // допустимо -- присваивание неконстантного значения константному
```

Тем не менее, такие присваивания перестают быть безопасными, когда вы переходите к двум уровням косвенности. Например, вы могли бы написать такой код:

```
const int **pp2;
int *p1;
const int n = 13;
pp2 = &p1;  // разрешено, но квалификатор const игнорируется
*pp2 = &n;  // допустимо, оба const, но p1 устанавливается указывающим на n
*p1 = 10;   // допустимо, но производится попытка изменить константу n
```

Что происходит? Как упоминалось ранее, в стандарте говорится, что результат изменения константных данных с использованием указателя, отличного от const, не определен. Например, компиляция короткой программы с этим кодом с помощью gcc в среде Terminal (интерфейс OS X для доступа к лежащей в основе системе Unix) приводит к тому, что n получает значение 13, но применение компилятора clang в той же среде обеспечивает для n значение 10. При этом оба компилятора предупреждают о несовместимых типах указателей. Разумеется, предупреждения можно игнорировать, но лучше не доверять результатам выполнения этой программы.

Квалификатор const в C и C++

В C и C++ квалификатор const используется похожим, но не идентичным образом. Одно из отличий состоит в том, что C++ позволяет применять целочисленное значение const для объявления размера массива, в то время как язык C является более ограничивающим. Второе отличие заключается в том, что язык C++ обладает более строгими правилами присваивания указателей:

```
const int y;
const int * p2 = &y;
int * p1;
p1 = p2;  // ошибка в C++, возможное предупреждение в C
```

В C++ не разрешено присваивать указатель const указателю, не являющемуся const. В C это присваивание возможно, но попытка использования p1 для изменения y ведет к неопределенному поведению.

Функции и многомерные массивы

Если вы намерены создавать функции, которые обрабатывают двумерные массивы, то должны достаточно хорошо понимать указатели, чтобы делать подходящие объявления для аргументов функций. В самом теле функции обычно можно обойтись записью в виде массива.

Давайте напишем функцию для взаимодействия с двумерными массивами. Одна из возможностей предусматривает использование цикла for для применения функции обработки одномерных массивов к каждой строке двумерного массива. Другими словами, можно предпринять примерно такие действия:

```
int junk[3][4] = { {2,4,5,8}, {3,5,6,9}, {12,10,8,6} };
int i, j;
int total = 0;
for (i = 0; i < 3 ; i++)
   total += sum(junk[i], 4);  // junk[i] - одномерный массив
```

Вспомните, что если junk — это двумерный массив, то junk[i] — одномерный массив, который можно рассматривать как одну строку в двумерном массиве. В таком случае функция sum() вычисляет промежуточную сумму для каждой строки двумерного массива, а в цикле for выполняется сложение этих промежуточных сумм.

Однако при таком подходе теряется возможность отслеживания информации о строках и столбцах. В этом приложении (суммирование всех значений) данная информация не является важной, но предположим, что каждая строка представляет год, а каждый столбец — месяц. Тогда вам может понадобиться функция, которая суммирует значения в отдельных столбцах. В таком случае функция должна располагать информацией о столбцах и строках. Этого можно достичь, объявив формальный параметр правильного вида, чтобы в функцию можно было корректно передавать массивы. В данной ситуации junk является массивом из трех массивов, содержащих по четыре элемента int. Как обсуждалось ранее, это означает, что junk представляет собой указатель на массив из четырех значений int. Параметр функции такого типа можно объявить следующим образом:

```
void somefunction( int (* pt)[4] );
```

В качестве альтернативы, если (и только если) pt является формальным параметром функции, его можно объявить так:

```
void somefunction( int pt[][4] );
```

Обратите внимание, что первая пара квадратных скобок пуста. Пустые квадратные скобки идентифицируют pt в качестве указателя. Затем переменная подобного рода может использоваться тем же способом, что и junk. Именно это сделано в следующем примере, показанном в листинге 10.17. В листинге демонстрируются три эквивалентных формы синтаксиса прототипов.

Листинг 10.17. Программа array2d.c

```
// array2d.c -- функции для двумерных массивов
#include <stdio.h>
#define ROWS 3
#define COLS 4

void sum_rows(int ar[][COLS], int rows);
void sum_cols(int [][COLS], int );          // имена можно опустить
int sum2d(int (*ar)[COLS], int rows);       // другой синтаксис
int main(void)
{
   int junk[ROWS][COLS] = {
         {2,4,6,8},
         {3,5,7,9},
         {12,10,8,6}
   };

   sum_rows(junk, ROWS);
   sum_cols(junk, ROWS);
   printf("Сумма всех элементов = %d\n", sum2d(junk, ROWS));

   return 0;
}

void sum_rows(int ar[][COLS], int rows)
{
   int r;
   int c;
   int tot;

   for (r = 0; r < rows; r++)
   {
      tot = 0;
      for (c = 0; c < COLS; c++)
         tot += ar[r][c];
      printf("строка %d: сумма = %d\n", r, tot);
   }
}

void sum_cols(int ar[][COLS], int rows)
{
   int r;
   int c;
   int tot;

   for (c = 0; c < COLS; c++)
   {
      tot = 0;
      for (r = 0; r < rows; r++)
      tot += ar[r][c];
      printf("столбец %d: сумма = %d\n", c, tot);
   }
}
```

```
int sum2d(int ar[][COLS], int rows)
{
   int r;
   int c;
   int tot = 0;

   for (r = 0; r < rows; r++)
      for (c = 0; c < COLS; c++)
         tot += ar[r][c];

   return tot;
}
```

Вот как выглядит вывод:

```
строка 0: сумма = 20
строка 1: сумма = 24
строка 2: сумма = 36
столбец 0: сумма = 17
столбец 1: сумма = 19
столбец 2: сумма = 21
столбец 3: сумма = 23
Сумма всех элементов = 80
```

Программа из листинга 10.17 передает функциям в качестве аргументов имя junk, которое является указателем на первый элемент, т.е. подмассив, и символическую константу ROWS, представляющую значение 3, т.е. количество строк. Затем каждая функция трактует ar как массив массивов, содержащих по четыре значения int. Количество столбцов встроено в каждую функцию, но количество строк остается незаданным. Те же самые функции будут работать, скажем, с массивом 12×4, если для количества строк передать число 12. Дело в том, что rows — это количество элементов, но поскольку каждый элемент является массивом, или строкой, rows превращается в количество строк.

Обратите внимание, что ar применяется в той же манере, как junk в функции main(). Это возможно потому, что ar и junk имеют одинаковый тип: указатель на массив из четырех значений int.

Имейте в виду, что следующее объявление не будет работать должным образом:

```
int sum2(int ar[][], int rows);    // ошибочное объявление
```

Вспомните, что компилятор переводит форму записи с массивами, в форму записи в стиле указателей. Это означает, например, что ar[1] превращается в ar+1. Чтобы компилятор мог оценить такое выражение, он должен знать размер объекта, на который указывает ar. Объявление

```
int sum2(int ar[][4], int rows);   // допустимое объявление
```

говорит о том, что ar указывает на массив из четырех значений int (следовательно, на объект длиной 16 байтов в нашей системе), поэтому ar+1 означает "добавить 16 байтов к адресу". В версии с пустыми квадратными скобками компилятор не будет знать, что делать дальше.

Можно также включить размер в другую пару квадратных скобок, как показано ниже, но компилятор его проигнорирует:

```
int sum2(int ar[3][4], int rows); // допустимое объявление, 3 игнорируется
```

Это удобно при использовании typedef (как упоминалось в главе 5 и будет описано в главе 14):

```
typedef int arr4[4];                // массив arr4 из 4 значений int
typedef arr4 arr3x4[3];             // массив arr3x4 из 3 массивов arr4
int sum2(arr3x4 ar, int rows);      // то же, что и следующее объявление
int sum2(int ar[3][4], int rows);   // то же, что и следующее объявление
int sum2(int ar[][4], int rows);    // стандартная форма
```

В общем случае, чтобы объявить указатель, соответствующий *N*-мерному массиву, вы должны задать значения во всех парах квадратных скобок, кроме самой левой:

```
int sum4d(int ar[][12][20][30], int rows);
```

Причина в том, что первый комплект квадратных скобок указывает на объявление именно указателя, тогда как остальные квадратные скобки описывают типы объектов данных, на которые ссылается указатель, как демонстрируется в следующем эквивалентном прототипе:

```
int sum4d(int (*ar)[12][20][30], int rows); // ar - указатель
```

Здесь `ar` указывает на массив размером 12×20×30 со значениями типа `int`.

Массивы переменной длины

Вы могли уже заметить странность в функциях, имеющих дело с двумерными массивами: количество строк можно описать с помощью параметра функции, но количество столбцов встроено в функцию. Например, взгляните на такое определение:

```
#define COLS 4
int sum2d(int ar[][COLS], int rows)
{
   int r;
   int c;
   int tot = 0;
   for (r = 0; r < rows; r++)
      for (c = 0; c < COLS; c++)
         tot += ar[r][c];
   return tot;
}
```

Предположим, что были объявлены следующие массивы:

```
int array1[5][4];
int array2[100][4];
int array3[2][4];
```

Функцию `sum2d()` можно применять с любым из этих массивов:

```
tot = sum2d(array1, 5);   // суммирование элементов массива 5 x 4
tot = sum2d(array2, 100); // суммирование элементов массива 100 x 4
tot = sum2d(array3, 2);   // суммирование элементов массива 2 x 4
```

Это объясняется тем, что количество строк передается в параметре `rows`, представляющем собой переменную. Однако если нужно просуммировать массив размером 6×5, придется использовать новую функцию, в которой параметр `COLS` должен быть определен как 5. Такое поведение является результатом того факта, что для указания размерности массива необходимо применять константы; таким образом, `COLS` нельзя заменить переменной. Если вы действительно хотите создать одну функцию, которая будет работать с двумерным массивом любого размера, это можно сделать, но придется приложить немало труда. (Вы должны передать массив как одномерный и заставить функцию вычислять, где начинается каждая строка.) Более того, этот прием недоста-

точно гладко сочетается с подпрограммами на языке FORTRAN, которые позволяют указывать в вызове функции оба измерения. Конечно, FORTRAN можно считать древним языком программирования, но за прошедшие десятилетия эксперты в области численных методов разработали на FORTRAN много удобных библиотек вычислений. Язык С позиционируется как наследник FORTRAN, поэтому возможность преобразования библиотек FORTRAN с минимальными усилиями является весьма полезной.

Такая потребность стала главным мотивом для ввода в стандарт С99 понятия массивов переменной длины, которые позволяют использовать переменные для размерности массива. Например, можно поступить так:

```
int quarters = 4;
int regions = 5;
double sales[regions][quarters];  // массив переменной длины
```

Как упоминалось ранее, с массивами переменной длины связаны некоторые ограничения. Они должны иметь автоматический класс хранения, а это означает их объявление либо в функции без применения модификаторов класса хранения `static` или `extern` (глава 12), либо в виде параметров функции. Кроме того, инициализация в объявлении невозможна. Наконец, в стандарте C11 массивы переменной длины являются необязательной возможностью языка в отличие от C99, где они были обязательными.

НА ЗАМЕТКУ! Массивы переменной длины не изменяют размер

Понятие *переменный* в массиве переменной длины вовсе не означает возможность изменения длины массива после его создания. Будучи созданным, массив переменной длины сохраняет тот же самый размер. В действительности понятие *переменный* означает, что при указании размерностей при первоначальном создании массива можно использовать переменные.

Поскольку массивы переменной длины представляют собой новое дополнение языка, их поддержка к настоящему времени пока не завершена. Давайте рассмотрим простой пример, в котором показано, как написать функцию для суммирования содержимого любого двумерного массива значений типа `int`. Для начала вот объявление функции с аргументом в виде двумерного массива переменной длины:

```
int sum2d(int rows, int cols, int ar[rows][cols]); //ar - массив переменной длины
```

Обратите внимание на то, что два первых параметра (`rows` и `cols`) применяются в качестве размерностей для объявления параметра типа массива `ar`. Поскольку в объявлении `ar` используются `rows` и `cols`, в списке параметров они должны быть объявлены до появления `ar`. Поэтому следующий прототип является ошибочным:

```
int sum2d(int ar[rows][cols], int rows, int cols);  // некорректный порядок
```

В стандартах C99/C11 утверждается, что имена в прототипе можно опускать, но в этом случае размерности должны быть заменены звездочками:

```
int sum2d(int, int, int ar[*][*]); //ar - массив переменной длины, имена не указаны
```

Теперь посмотрите, как определять функцию:

```
int sum2d(int rows, int cols, int ar[rows][cols])
{
   int r;
   int c;
   int tot = 0;
   for (r = 0; r < rows; r++)
      for (c = 0; c < cols; c++)
         tot += ar[r][c];
   return tot;
}
```

Если не считать нового заголовка функции, то единственное отличие этой функции от ее классической версии на С (листинг 10.17) состоит в том, что константа COLS была заменена переменной cols. Такое изменение стало возможным благодаря присутствию в заголовке функции массива переменной длины. Кроме того, наличие переменных, которые представляют количество строк и столбцов, позволяет применять новую функцию sum2d() с любым размером двумерного массива значений int. Данное утверждение иллюстрируется в листинге 10.18. Однако для компиляции такого кода требуется компилятор С, в котором реализована поддержка массивов переменной длины. Здесь также демонстрируется возможность использования этой функции, основанной на массиве переменной длины, либо с традиционными массивами С, либо с массивом переменной длины.

Листинг 10.18. Программа vararr2d.c

```
//vararr2d.c -- функции, использующие массивы переменной длины
#include <stdio.h>
#define ROWS 3
#define COLS 4
int sum2d(int rows, int cols, int ar[rows][cols]);
int main(void)
{
   int i, j;
   int rs = 3;
   int cs = 10;
   int junk[ROWS][COLS] = {
      {2,4,6,8},
      {3,5,7,9},
      {12,10,8,6}
   };
   int morejunk[ROWS-1][COLS+2] = {
      {20,30,40,50,60,70},
      {5,6,7,8,9,10}
   };
   int varr[rs][cs]; // массив переменной длины
   for (i = 0; i < rs; i++)
      for (j = 0; j < cs; j++)
         varr[i][j] = i * j + j;
   printf("Традиционный массив 3x5\n");
   printf("Сумма всех элементов = %d\n",
         sum2d(ROWS, COLS, junk));
   printf("Традиционный массив 2x6\n");
   printf("Сумма всех элементов = %d\n",
         sum2d(ROWS-1, COLS+2, morejunk));
   printf("Массив переменной длины 3x10\n");
   printf("Сумма всех элементов = %d\n",
         sum2d(rs, cs, varr));
   return 0;
}
// функция с параметром типа массива переменной длины
int sum2d(int rows, int cols, int ar[rows][cols])
{
   int r;
   int c;
   int tot = 0;
   for (r = 0; r < rows; r++)
      for (c = 0; c < cols; c++)
         tot += ar[r][c];
   return tot;
}
```

Ниже приведен вывод, полученный из этой программы:

```
Традиционный массив 3x5
Сумма всех элементов = 80
Традиционный массив 2x6
Сумма всех элементов = 315
Массив переменной длины 3x10
Сумма всех элементов = 270
```

Следует отметить, что объявление массива переменной длины в списке параметров определения функции в действительности не приводит к созданию массива. Как в старом синтаксисе, имя массива переменной длины на самом деле является указателем. Это значит, что функция с параметром в виде массива переменной длины фактически работает с данными в исходном массиве и потому имеет возможность модифицировать массив, переданный в качестве аргумента. В следующем фрагменте кода показано, когда объявляется указатель, а когда — действительный массив:

```
    int thing[10][6];
    twoset(10,6,thing);
    ...
}
void twoset (int n, int m, int ar[n][m])     // ar - это указатель на
                                             // массив из m значений int
{
    int temp[n][m];   // temp - массив n x m значений int
    temp[0][0] = 2;   // установка значения элемента массива temp в 2
    ar[0][0] = 2;     // установка значения thing[0][0] в 2
}
```

Когда функция `twoset()` вызывается, как выше в коде, `ar` становится указателем на `thing[0]`, а `temp` создается как массив 10×6. Поскольку `ar` и `thing` являются указателями на `thing[0]`, то `ar[0][0]` обращается к тому же самому месту в данных, что и `thing[0][0]`.

Массивы переменной длины также делают возможным динамическое выделение памяти. Это означает, что размер массива можно указывать во время выполнения программы. К обычным массивам С применяется статическое выделение памяти, предполагающее, что размер массив определяется на этапе компиляции. Причина в том, что размеры массива, будучи константами, известны компилятору. Динамическое выделение памяти обсуждается в главе 12.

const и размеры массивов

Можно ли использовать символическую константу, определенную с помощью `const`, при объявлении массива?

```
const int SZ = 80;
...
double ar[SZ];  // разрешено?
```

Для стандарта С90 ответ отрицателен (скорее всего). Размер должен быть задан целочисленным константным выражением, которое может быть комбинацией целочисленных констант, таких как `20`, выражений `sizeof` и нескольких других элементов, ни один из которых не имеет отношения к `const`. Конкретная реализация может расширять диапазон того, что считается целочисленным константным выражением, тогда можно будет применять символические константы `const`, но такой код перестанет быть переносимым.

Для стандартов С99/С11 ответ положителен, если массив иначе мог бы быть массивом переменной длины. Следовательно, определение должно делаться для массива с автоматическим классом хранения, объявленного внутри блока.

Составные литералы

Предположим, что вы хотите передать в функцию значение с помощью параметра int; вы можете передать переменную int, но также константу int, такую как 5. До выхода стандарта C99 положение дел с функцией, принимающей аргумент типа массива, было другим; можно было передавать массив, но отсутствовал эквивалент для константы типа массива. Стандарт C99 изменил эту ситуацию, введя *составные литералы*. Литералы — это константы, которые не являются символическими.

Например, 5 — литерал типа int, 81.3 — литерал типа double, 'Y' — литерал типа char, а "elephant" — строковый литерал. В комитете, разрабатывающем стандарт C99, пришли к соглашению, что было бы удобно иметь составные литералы, которые могли бы представлять содержимое массивов и структур.

Для массивов составной литерал выглядит подобно списку инициализации массива, который предварен именем типа, заключенным в круглые скобки. Например, вот обычное объявление массива:

```
int diva[2] = {10, 20};
```

А вот составной литерал, который создает неименованный массив, содержащий те же два значения int:

```
(int [2]){10, 20}    // составной литерал
```

Обратите внимание, что имя типа — это то, что остается после удаления diva из предыдущего объявления, т.е. int [2].

Точно так же, как размер массива можно не указывать во время инициализации именованного массива, его можно опускать в составном литерале, и компилятор подсчитает количество присутствующих элементов:

```
(int []){50, 20, 90} // составной литерал с тремя элементами
```

Поскольку эти составные литералы не имеют имени, нельзя просто создать их в одном операторе и затем использовать их позже. Вместо этого они должны каким-то образом применяться при создании. Один из способов предусматривает использование указателя для отслеживания ячейки памяти. То есть вы можете поступить примерно так:

```
int * pt1;
pt1 = (int [2]) {10, 20};
```

Как видите, эта литеральная константа идентифицируется как массив значений int. Подобно имени массива эта константа транслируется в адрес первого элемента, поэтому ее можно присвоить указателю на тип int. После этого указатель можно применять далее в коде. Например, *pt1 в данном случае будет иметь значение 10, а pt1[1] — 20.

Еще одним возможным действием с составным литералом будет его передача в качестве фактического аргумента функции с совпадающим формальным параметром:

```
int sum(const int ar[], int n);
...
int total3;
total3 = sum((int []){4,4,4,5,5,5}, 6);
```

Здесь первый аргумент представляет собой массив из шести элементов int, который действует как адрес первого элемента, т.е. так же, как имя массива. Такой вид использования, при котором информация передается функции без необходимости заранее создавать массив, типичен для составных литералов.

Этот прием можно распространить на двумерные и многомерные массивы. Ниже приведен пример создания двумерного массива значений int и сохранения его адреса:

```
int (*pt2)[4]; // объявление указателя на массив из массивов с 4 значениями int
pt2 = (int [2][4]) { {1,2,3,-9}, {4,5,6,-8} };
```

В данном случае типом является int [2][4] — массив 2×4 значений int.

В листинге 10.19 все эти примеры объединены в одну завершенную программу.

Листинг 10.19. Программа flc.c

```
// flc.c -- забавно выглядящие константы
#include <stdio.h>
#define COLS 4
int sum2d(const int ar[][COLS], int rows);
int sum(const int ar[], int n);
int main(void)
{
   int total1, total2, total3;
   int * pt1;
   int (*pt2)[COLS];

   pt1 = (int [2]) {10, 20};
   pt2 = (int [2][COLS]) { {1,2,3,-9}, {4,5,6,-8} };

   total1 = sum(pt1, 2);
   total2 = sum2d(pt2, 2);
   total3 = sum((int []){4,4,4,5,5,5}, 6);
   printf("total1 = %d\n", total1);
   printf("total2 = %d\n", total2);
   printf("total3 = %d\n", total3);

   return 0;
}

int sum(const int ar[], int n)
{
   int i;
   int total = 0;

   for( i = 0; i < n; i++)
      total += ar[i];

   return total;
}
int sum2d(const int ar[][COLS], int rows)
{
   int r;
   int c;
   int tot = 0;

   for (r = 0; r < rows; r++)
      for (c = 0; c < COLS; c++)
         tot += ar[r][c];
   return tot;
}
```

Вам понадобится компилятор, принимающий данное дополнение стандарта C99 (не все компиляторы делают это). Вот вывод этой программы:

```
total1 = 30
total2 = 4
total3 = 27
```

Имейте в виду, что составной литерал — это средство для предоставления значений, которые нужны лишь временно. Его область действия ограничена блоком, как будет прояснено в главе 12. Это значит, что его существование не гарантируется после того, как поток выполнения программы покинет блок, в котором определен составной литерал, т.е. самую внутреннюю пару фигурных скобок, содержащую определение.

Ключевые понятия

Когда возникает необходимость хранить множество значений одного вида, решением может быть массив. В языке С массивы считаются *производными типами*, поскольку они построены на основе других типов. Другими словами, вы объявляете не просто массив, а массив значений `int`, `float` или какого-то другого типа. Такой тип сам по себе может быть типом массива, что дает в итоге массив из массивов, или двумерный массив.

Часто полезно создавать функции для обработки массивов; это содействует модульности программы за счет размещения специфических задач в специализированных функциях. Важно понимать, что когда вы используете имя массива в качестве фактического аргумента, то функции не передается весь массив целиком, а только его адрес (следовательно, соответствующий формальный параметр является указателем). Чтобы обработать этот массив, функция должна знать, где хранится массив и сколько элементов он содержит. Адрес массива сообщает о том, где он находится, а данные о количестве элементов должны либо быть встроенными в функцию, либо передаваться ей в отдельном аргументе. Второй подход более универсален, поэтому одна и та же функция может работать с массивами разных размеров.

Связь между массивами и указателями настолько тесная, что часто одну и ту же операцию можно представить с применением либо формы записи с массивами, либо формы записи с указателями. Именно эта связь позволяет использовать внутри функции, обрабатывающей массивы, форму записи с массивами даже в случае, когда формальный параметр является указателем, а не массивом.

В языке С размер обычного массива должен быть задан с помощью константного выражения, поэтому он определяется на этапе компиляции. Стандарты C99/C11 предлагают альтернативу в виде массивов переменной длины, когда спецификатором размера может быть переменная. Это позволяет откладывать установку размера массива переменной длины до времени выполнения программы.

Резюме

Массив — это набор элементов, которые имеют один и тот же тип данных. Элементы массива хранятся в памяти последовательно, а доступ к ним осуществляется с применением целочисленного индекса (или *смещения*). В языке С первый элемент массива имеет индекс `0`, поэтому последний элемент в массиве из `n` элементов имеет индекс `n - 1`. Ответственность за обеспечение допустимости используемых индексов возлагается на программиста, поскольку ни компилятор, ни выполняющаяся программа не должны ее проверять.

Для объявления простого *одномерного* массива применяется следующая форма:

```
тип имя[размер];
```

Здесь *тип* — это тип данных каждого элемента массива, *имя* — имя массива, а *размер* — количество элементов. Традиционно язык С требовал, чтобы *размер* был константным целочисленным выражением. Стандарт C99/C11/C11 разрешает ис-

пользовать неконстантное целочисленное выражение; в таком случае массив называется массивом переменной длины.

В С имя массива интерпретируется как адрес первого элемента этого массива. Другими словами, имя массива эквивалентно указателю на его первый элемент. В целом массивы и указатели тесно связаны друг с другом. Если ar — это массив, то выражения ar[i] и *(ar + i) эквивалентны.

Язык С не позволяет передавать весь массив целиком в качестве аргумента функции, но можно передать адрес массива. Функция затем может применять этот адрес для манипулирования исходным массивом. Если функция не предназначена для модификации исходного массива, то при объявлении формального параметра, представляющего массив, должно использоваться ключевое слово const. Внутри функции можно применять либо форму записи в виде массивов, либо форму записи в виде указателей. В любом случае на самом деле используется переменная типа указателя.

Добавление к указателю целого числа или инкрементирование указателя изменяет его значение на количество байтов, занимаемое в памяти объектом, на который ссылается указатель. То есть если pd указывает на 8-байтовое значение типа double в массиве, то добавление 1 к указателю pd увеличивает его значение на 8, так что указатель будет ссылаться на следующий элемент массива.

Двумерные массивы представляют массивы массивов. Например, объявление

```
double sales[5][12];
```

создает массив по имени sales, имеющий пять элементов, каждый из которых является массивом из 12 значений типа double. На первый из этих одномерных массивов можно ссылаться как на sales[0], на второй — sales[1] и т.д., причем каждый из этих массивов содержит 12 значений double. Второй индекс служит для доступа к конкретным элементам в этих массивах. Например, sales[2][5] — это шестой элемент массива sales[2], а sales[2] — третий элемент массива sales.

Традиционный для С метод передачи многомерного массива в функцию заключается в передаче имени массива, которое является адресом, параметру подходящего типа указателя. Объявление такого указателя должно описывать все размерности массива кроме первой; размерность первого параметра обычно передается во втором аргументе. Например, чтобы обработать ранее упоминавшийся массив sales, прототип функции и вызов функции должны иметь вид:

```
void display(double ar[][12], int rows);
...
display(sales, 5);
```

Массивы переменной длины предоставляют второй синтаксис, при котором обе размерности передаются функции в качестве аргументов. В этом случае прототип функции и вызов функции выглядят так:

```
void display(int rows, int cols, double ar[rows][cols]);
...
display(5, 12, sales);
```

В обсуждении участвовали массивы значений типа int и типа double, но те же концепции применимы к массивам других типов. Тем не менее, в отношении символьных строк действует много специальных правил. Это вытекает из того факта, что завершающий нулевой символ в строке предоставляет функциям способ обнаружения конца строки без необходимости в передаче им размера. Символьные строки будут подробно рассматриваться в главе 11.

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля приведены в приложении А.

1. Что выведет следующая программа?

```
#include <stdio.h>
int main(void)
{
    int ref[] = {8, 4, 0, 2};
    int *ptr;
    int index;

    for (index = 0, ptr = ref; index < 4; index++, ptr++)
        printf("%d %d\n", ref[index], *ptr);
    return 0;
}
```

2. Сколько элементов содержит массив `ref` из вопроса 1?
3. Адресом чего является `ref` из вопроса 1? Что можно сказать о `ref + 1`? На что указывает `++ref`?
4. Каковы значения `*ptr` и `*(ptr + 2)` в каждом из следующих случаев?

а.
```
int *ptr;
int torf[2][2] = {12, 14, 16};
ptr = torf[0];
```

б.
```
int * ptr;
int fort[2][2] = { {12}, {14,16} };
ptr = fort[0];
```

5. Каковы значения `**ptr` и `**(ptr + 1)` в каждом из следующих случаев?

а.
```
int (*ptr)[2];
int torf[2][2] = {12, 14, 16};
ptr = torf;
```

б.
```
int (*ptr)[2];
int fort[2][2] = { {12}, {14,16} };
ptr = fort;
```

6. Предположим, что имеется следующее объявление:

```
int grid[30][100];
```

а. Выразите адрес `grid[22][56]` одним способом.

б. Выразите адрес `grid[22][0]` двумя способами.

в. Выразите адрес `grid[0][0]` тремя способами.

7. Создайте подходящее объявление для каждой из описанных ниже переменных.

а. `digits` представляет собой массив из 10 значений `int`.

б. `rates` представляет собой массив из шести значений `float`.

в. `mat` представляет собой массив, состоящий из трех массивов, каждый из которых содержит 5 целых чисел.

г. `psa` представляет собой массив, состоящий из 20 указателей на `char`.

д. `pstr` представляет собой указатель на массив, состоящий из 20 значений `char`.

8. Решите перечисленные ниже задачи.
   а. Объявите массив, состоящий из шести значений типа `int`, и инициализируйте его значениями 1, 2, 4, 8, 16 и 32.
   б. Используйте форму запись с массивом для представления третьего элемента (имеющего значение 4) массива, объявленного в пункте а).
   в. Предполагая, что действуют правила C99/C11, объявите массив из 100 значений типа `int` и инициализируйте его таким образом, чтобы последний элемент получил значение -1; значения остальных элементов могут быть произвольными.
   г. Предполагая, что действуют правила C99/C11, объявите массив из 100 значений типа int и инициализируйте его так, чтобы элементы 5, 10, 11, 12 и 3 получили значение 101; значения остальных элементов могут быть произвольными.
9. Каков диапазон значений индекса в 10-элементном массиве?
10. Предположим, что имеются следующие объявления:

```
float rootbeer[10], things[10][5], *pf, value = 2.2;
int i = 3;
```

Укажите, какие из приведенных ниже операторов допустимы, а какие — нет:

а. `rootbeer[2] = value;`
б. `scanf("%f", &rootbeer);`
в. `rootbeer = value;`
г. `printf("%f", rootbeer);`
д. `things[4][4] = rootbeer[3];`
е. `things[5] = rootbeer;`
ж. `pf = value;`
з. `pf = rootbeer;`

11. Объявите массив размерности 800×600 значений типа `int`.
12. Имеются три объявления массивов:

```
double trots[20];
short clops[10][30];
long shots[5][10][15];
```

   а. Напишите прототип и оператор вызова для традиционной функции типа void, которая обрабатывает массив `trots`, и для функции C, использующей массив переменной длины.
   б. Напишите прототип и оператор вызова для традиционной функции типа void, которая обрабатывает массив `clops`, и для функции, использующей массив переменной длины.
   в. Напишите прототип и оператор вызова для традиционной функции типа void, которая обрабатывает массива `shots`, и для функции, использующей массив переменной длины.
13. Имеются два прототипа функций:

```
void show(const double ar[], int n);          // n - количество элементов
void show2(const double ar2[][3], int n);     // n - количество строк
```

   а. Напишите вызов функции, который передает `show()` составной литерал, содержащий значения 8, 3, 9 и 2.
   б. Напишите вызов функции, который передает `show2()` составной литерал, содержащий значения 8, 3 и 9 в первой строке и значения 5, 4 и 1 во второй строке.

Упражнения по программированию

1. Модифицируйте программу `rain.c` из листинга 10.7, чтобы она выполняла вычисления с использованием указателей вместо индексов. (Вам по-прежнему придется объявлять и инициализировать массив.)

2. Напишите программу, которая инициализирует массив значений типа `double` и затем копирует его содержимое в три других массива. (Все четыре массива должны быть объявлены в главной программе.) Для создания первой копии воспользуйтесь функцией, в которой применяется форма записи с массивами. Для создания второй копии используйте функцию, в которой применяется форма записи с указателями и инкрементирование указателей. Первые две функции должны принимать в качестве аргументов имя целевого массива, имя исходного массива и количество элементов, подлежащих копированию. Третья функция должна принимать в качестве аргументов имя целевого массива, имя исходного массива и указатель на элемент, следующий за последним элементом в исходном массиве. С учетом приведенных ниже объявлений вызовы функций должны выглядеть так:

```
double source[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
double target1[5];
double target2[5];
double target3[5];

copy_arr(target1, source, 5);
copy_ptr(target2, source, 5);
copy_ptrs(target3, source, source + 5);
```

3. Напишите функцию, которая возвращает наибольшее значение в массиве значений типа `int`. Протестируйте эту функцию с помощью простой программы.

4. Напишите функцию, которая возвращает индекс наибольшего значения в массиве значений типа `double`. Протестируйте эту функцию с помощью простой программы.

5. Напишите функцию, которая возвращает разность между наибольшим и наименьшим элементами в массиве значений типа `double`. Протестируйте эту функцию с помощью простой программы.

6. Напишите функцию, которая изменяет порядок следования содержимого массива значений типа `double` на противоположный и протестируйте ее в простой программе.

7. Напишите программу, которая инициализирует двумерный массив значений типа `double` и использует одну из функций копирования из упражнения 2 для его копирования во второй двумерный массив. (Поскольку двумерный массив – это массив массивов, функция для копирования одномерных массивов может применяться с каждым подмассивом.)

8. Воспользуйтесь одной из функций копирования из упражнения 2 для копирования элементов с 3-го по 5-й семиэлементного массива в массив, состоящий из трех элементов. Саму функцию изменять не нужно; просто подберите правильные фактические аргументы. (Фактическими аргументами не обязательно должны быть имя массива и его размер. Ими только должны быть адрес элемента массива и количество обрабатываемых элементов.)

9. Напишите программу, которая инициализирует двумерный массив 3 × 5 значений типа double и использует функцию, основанную на массивах переменной длины, для копирования этого массива во второй двумерный массив. Кроме того, напишите функцию, основанную на массивах переменной длины, для отображения содержимого этих двух массивов. В общем случае обе функции должны быть способны обрабатывать произвольные массивы $N \times M$. (Если вы не имеете доступа к компилятору, поддерживающему массивы переменной длины, примените традиционный подход C с функциями, которые могут обрабатывать массивы $N \times 5$).

10. Напишите функцию, которая устанавливает значение каждого элемента массива в сумму соответствующих элементов в двух других массивах. Другими словами, если массив 1 имеет значения 2, 4, 5 и 8, а массив 2 – значения 1, 0, 4 и 6, то эта функция присваивает массиву 3 значения 3, 4, 9 и 14. Функция должна принимать в качестве аргументов имена трех массивов и их размеры. Протестируйте эту функцию с помощью простой программы.

11. Напишите программу, которая объявляет массив 3×5 значений типа int и инициализирует его значениями по вашему выбору. Программа должна вывести эти значения, удвоить все значения, после чего вывести новые значения. Напишите одну функцию для вывода значений и еще одну для удваивания значений. В качестве аргументов функции должны принимать имя массива и количество строк.

12. Перепишите программу rain.c из листинга 10.7 так, чтобы основные задачи решались внутри функций, а не в теле main().

13. Напишите программу, которая предлагает пользователю ввести три набора по пять чисел double. (Для простоты можете полагать, что пользователь отвечает корректно и не вводит нечисловые данные.) Программа должна выполнять все перечисленные ниже действия.

    а. Хранить информацию в массиве 3×5.

    б. Вычислять среднее для каждого набора из пяти значений.

    в. Вычислять среднее для всех значений.

    г. Определять наибольшее из 15 значений.

    д. Выводить результаты на экран.

    Каждая значительная задача должна решаться посредством отдельной функции с использованием традиционного в языке C подхода к обработке массивов. Выполните задачу б) с помощью функции, которая вычисляет и возвращает среднее значение одномерного массива; воспользуйтесь циклом для вызова этой функции три раза. Функции, реализующие остальные задачи, должны принимать в качестве аргумента массив целиком, а функции, выполняющие задачи в) и г), должны возвращать ответ в вызывающую программу.

14. Выполните упражнение 13, но используйте в качестве параметров функции массивы переменной длины.

11

Символьные строки и строковые функции

В ЭТОЙ ГЛАВЕ...

- Функции: gets(), gets_s(), fgets(), puts(), fputs(), strcat(), strncat(), strcmp(), strncmp(), strcpy(), strncpy(), sprintf(), strchr()
- Создание и использование строк
- Применение строковых и символьных функций из библиотеки C и создание собственных строковых функций
- Использование аргументов командной строки

Символьная строка является одним из наиболее полезных и важных типов данных в языке С. До сих пор мы постоянно использовали символьные строки, но вам еще предстоит многое узнать о них. Стандартная библиотека С предлагает широкий спектр функций для чтения и записи, копирования, сравнения, объединения, поиска и выполнения многих других операций со строками. Эта глава поможет пополнить свои навыки программирования возможностями, которые касаются строк.

Введение в строки и строковый ввод-вывод

Разумеется, вам уже известен наиболее важный факт: *символьная строка* — это массив элементов типа char с завершающим нулевым символом (\0). Следовательно, все, что вы изучили о массивах и указателях, касается и символьных строк. Однако в связи с широким применением символьных строк библиотека С предоставляет множество функций, специально предназначенных для работы со строками. В этой главе обсуждается природа строк, способы их объявления и инициализации, ввод и вывод строк в программах, а также манипуляции со строками.

Давайте рассмотрим короткую программу (листинг 11.1), которая иллюстрирует несколько способов представления строк.

Листинг 11.1. Программа strings1.c

```
// strings1.c
#include <stdio.h>
#define MSG "Я - символьная строковая константа."
#define MAXLENGTH 81
int main(void)
{
    char words[MAXLENGTH] = "Я являюсь строкой, хранящейся в массиве.";
    const char * pt1 = "Что-то указывает на меня.";
    puts("Вот несколько строк:");
    puts(MSG);
    puts(words);
    puts(pt1);
    words[14] = 'ф';
    puts(words);

    return 0;
}
```

Подобно printf(), функция puts() принадлежит семейству функций ввода-вывода stdio.h. Она отображает только строки и, в отличие от printf(), автоматически добавляет к выводимой строке символ новой строки. Ниже показан вывод этой программы:

```
Вот несколько строк:
Я - старомодная символьная строковая константа.
Я являюсь строкой, хранящейся в массиве.
Что-то указывает на меня.
Я являюсь строфой, хранящейся в массиве.
```

Вместо того чтобы анализировать листинг 11.1 строка за строкой, мы примем другой подход. Первым делом мы рассмотрим способы определения строки в программе. Затем мы выясним, как осуществляется чтение строки в программе. И, наконец, мы исследуем методы для вывода строк.

Определение строк в программе

Во время просмотра листинга 11.1 вы, скорее всего, заметили, что существует много способов определения строк. К основным методам относится использование строковых констант, массивов типа `char` и указателей на тип `char`. Программа должна обеспечить место для хранения строки, и эту тему мы также обсудим.

Символьные строковые литералы (строковые константы)

Строковый литерал, который также называют *строковой константой*, представляет собой произвольную последовательность символов, помещенную в двойные кавычки. Заключенные в кавычки символы, а также завершающий символ `\0`, который автоматически добавляется компилятором, хранятся в памяти как символьная строка. Таким образом, `"Я - символьная строковая константа."`, `"Я являюсь строкой, хранящейся в массиве."`, `"Что-то указывает на меня."` и `"Вот несколько строк:"` — все это строковые литералы.

Вспомните, что, начиная со стандарта ANSI C, выполняется конкатенация строковых литералов, если они отделены друг от друга ничем, кроме пробельных символов. Например, определение

```
char greeting[50] = "Здравствуйте, "" как вы себя" " чувствуете"
                    " сегодня?";
```

эквивалентно следующему определению:

```
char greeting[50] = "Здравствуйте, как вы себя чувствуете сегодня?";
```

Если вы хотите применить двойные кавычки в строке, предварите их обратной косой чертой:

```
printf("\"Беги, Спот, беги!\" - воскликнул Дик.\n");
```

Вывод будет таким:

```
"Беги, Спот, беги!" - воскликнул Дик.
```

Символьные строковые константы размещаются в *статическом классе хранения*, т.е. если вы используете строковую константу в функции, то эта строка сохраняется только однажды и существует на протяжении времени выполнения программы, даже если функция вызывается много раз. Вся фраза, заключенная в кавычки, действует в качестве указателя на место, где хранится строка. Это аналогично имени массива, которое трактуется как указатель на место размещения массива. Если сказанное верно, то какой вывод должна сгенерировать программа в листинге 11.2?

Листинг 11.2. Программа `strptr.c`

```
/* quotes.c -- строки как указатели */
#include <stdio.h>
int main(void)
{
   printf("%s, %p, %c\n", "Мы", " - ", *"космические бродяги");

   return 0;
}
```

Формат `%s` должен вывести строку `Мы`. Формат `%p` выводит адрес. Таким образом, если фраза `" - "` является адресом, то формат `%p` должен обеспечить вывод адреса первого символа в этой строке. (В реализациях, предшествующих стандарту ANSI C, может понадобиться заменить `%p` спецификатором `%u` или `%lu`.)

Наконец, выражение *"космические бродяги" должно дать значение, на которое указывает адрес и которым будет первый символ строки "космические бродяги". Так ли это на самом деле? Взглянем на вывод:

```
Мы, 0x100000f61, к
```

Массивы символьных строк и инициализация

Когда вы определяете массив символьных строк, то должны сообщить компилятору, сколько для него необходимо выделить памяти. Один из способов предусматривает указание размера массива, достаточного для хранения строки. Следующее объявление инициализирует массив m1 символами заданной строки:

```
const char m1[40] = "Постарайтесь уложиться в одну строку.";
```

Ключевое слово const отражает намерение не изменять эту строку.

Показанная форма является сокращением для стандартной формы инициализации массива:

```
const char m1[40] = { 'П',
'о', 'с', 'т', 'а', 'р', 'а', 'й', 'т', 'е', 'с', 'ь', ' ',
'у', 'л', 'о', 'ж', 'и', 'т', 'ь', 'с', 'я',
' ', 'в', ' ', 'о', 'д', 'н', 'у', ' ', 'с', 'т',
'р', 'о', 'к', 'у', '.', '\0'
};
```

Обратите внимание на завершающий нулевой символ. Без него вы получите символьный массив, а не строку.

При указании размера массива убедитесь, что количество элементов, по меньшей мере, на единицу больше длины строки (не забывайте о нулевом символе). Любые неиспользованные элементы инициализируются значением 0 (которое представляет собой нулевой символ в форме char). На рис. 11.1 приведена иллюстрация.

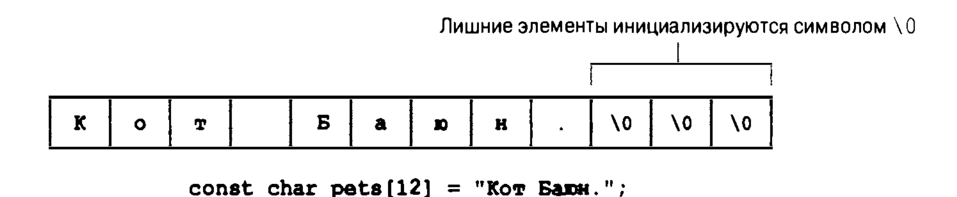


Рис. 11.1. Инициализация массива

Часто удобно предоставить определение размера массива компилятору; вспомните, что если размер не указан в инициализирующем объявлении, компилятор самостоятельно определит этот размер:

```
const char m2[] = "Если вам не о чем думать, вообразите что-нибудь.";
```

Инициализация символьных массивов — один из тех случаев, когда действительно имеет смысл позволить компилятору определять размер массива. Причина в том, что функции обработки массивов не нуждаются в знании размера массива, т.к. они могут просто находить нулевой символ, отмечающий конец.

Предоставление компилятору возможности вычислить размер массива самостоятельно работает только при инициализации массива. Если вы создаете массив, который намерены заполнить позже, его размер необходимо указывать в объявлении.

Внутри объявления размер массива должен вычисляться как целочисленное значение. До ввода стандартом C99 массивов переменной длины (variable length array – VLA) размер должен был быть целочисленной константой, что включает возможность применения выражения, образованного из константных целочисленных значений.

```
int n = 8;
char cookies[1];  //допустимо
char cakes[2 + 5]; //допустимо, поскольку размер является константным выражением
char pies[2*sizeof(long double) + 1];  // допустимо
char crumbs[n];   // не допускалось до выхода стандарта C99;
                  // в C99 это массив переменной длины
```

Подобно любому имени массива, имя символьного массива выдает адрес первого элемента массива. Следовательно, приведенное ниже утверждение справедливо:

```
char car[10] = "Луна";
car == &car[0], *car == 'Л' и *(car+1) == car[1] == 'у'
```

Действительно, для установки строки можно использовать форму записи с указателями. Например, в листинге 11.1 имеется следующее объявление:

```
const char * pt1 = "Что-то указывает на меня.";
```

Это объявление очень близко к такому объявлению:

```
const char ar1[] = "Что-то указывает на меня.";
```

Оба объявления говорят о том, что pt1 и ar1 являются адресами строк. В обоих случаях строка, заключенная в кавычки, сама определяет необходимый объем памяти, который будет для нее зарезервирован. Тем не менее, эти формы не идентичны.

Массивы или указатели

Так в чем же разница между формами в виде массива и указателя? Запись в форме массива (ar1[]) приводит к размещению в памяти компьютера массива из 26 элементов (по одному на каждый символ плюс один элемент для завершающего символа '\0'). Каждый элемент инициализируется соответствующим символом строкового литерала. Обычно строка, заключенная в кавычки, хранится в сегменте данных, который является частью исполняемого файла; когда программа загружается в память, вместе с ней загружается и эта строка. Говорят, что заключенная в кавычки строка находится в *статической памяти*. Однако память под массив выделяется только после того, как программа начнет выполнение. В это время строка, заключенная в кавычки, копируется в массив. (В главе 12 управление памятью обсуждается более подробно.) Обратите внимание, что в этот момент существуют две копии строки. Одна – это строковый литерал в статической памяти, а другая – строка, хранящаяся в массиве ar1.

В дальнейшем компилятор будет распознавать имя массива ar1 как синоним адреса первого элемента массива, &ar1[0]. Один важный аспект состоит здесь в том, что в форме массива ar1 является адресной *константой*. Значение ar1 изменять нельзя, т.к. это означало бы изменение места (адреса), где хранится массив. Для идентификации следующего элемента в массиве можно задействовать операции наподобие ar1+1, но выражение ++ar1 не разрешено. Операция инкремента может применяться только к именам переменных (или, в общем смысле, к модифицируемым l-значениям), но не к константам.

Форма указателя (*pt1) также приводит к тому, что в статической памяти под строку резервируются 26 элементов. Вдобавок, как только программа начнет выполняться, она выделяет в памяти место под *переменную* типа указателя pt1 и сохраняет в ней адрес строки. Первоначально эта переменная указывает на первый символ строки, но

ее значение можно изменять. Следовательно, можно использовать операцию инкремента. Например, ++pt1 будет указывать на второй символ ('т').

Строковые литералы считаются данными const. Поскольку указатель pt1 ссылается на такие данные, он должен быть объявлен, как указывающий на данные const. Это вовсе не означает, что нельзя изменять значение pt1 (т.е. место, на которое он указывает), просто pt1 не допускается применять для изменения самих данных. С другой стороны, при копировании строкового литерала в массив данные можно свободно изменять, если только сам массив не был объявлен как const.

Короче говоря, инициализация массива приводит к копированию строки из статической памяти в массив, тогда как инициализация указателя просто копирует адрес строки. Эти утверждения демонстрируются в листинге 11.3.

Листинг 11.3. Программа addresses.c

```
// addresses.c -- адреса строк
#define MSG "Я особенный."

#include <stdio.h>
int main()
{
    char ar[] = MSG;
    const char *pt = MSG;

    printf("адрес \"Я особенный.\": %p \n", "Я особенный.");
    printf("              адрес ar: %p\n", ar);
    printf("              адрес pt: %p\n", pt);
    printf("             адрес MSG: %p\n", MSG);
    printf("адрес \"Я особенный.\": %p \n", "Я особенный.");

    return 0;
}
```

Вот вывод, полученный в одной из систем:

```
адрес "Я особенный.": 0x100000f0c
            адрес ar: 0x7fff5fbff8c7
            адрес pt: 0x100000ee0
           адрес MSG: 0x100000ee0
адрес "Я особенный.": 0x100000f0c
```

О чем это свидетельствует? Во-первых, pt и MSG — это один и тот же адрес, но ar, как и было сказано, является другим адресом. Во-вторых, хотя литерал "Я особенный." встречается в операторах printf() дважды, компилятор использует одну область памяти, но с адресом, отличающимся от адреса MSG. Компилятору предоставляется свобода выбора сохранять литерал, который применяется более одного раза, в одном или нескольких местах. Другой компилятор мог бы представить все три экземпляра "Я особенный." в одной области памяти. В-третьих, часть памяти, использованная для статических данных, отличается от применяемой для динамической памяти, которая выделена под ar. Различны не только значения; этот конкретный компилятор использует даже разное количество битов для представления двух видов памяти.

Важны ли различия между представлениями строк в виде массива и указателя? Часто нет, однако это зависит от того, что именно вы пытаетесь сделать. Продолжим рассмотрение этих вопросов.

Различия между массивами и указателями

Давайте исследуем отличия между инициализацией символьного массива, предназначенного для хранения строки, и инициализацией указателя, который указывает на эту строку. (Под "указанием на строку" подразумевается указание на первый символ строки.) Например, взгляните на следующие два объявления:

```
char heart[] = "Я люблю Тилли!";
const char *head = "Я люблю Милли!";
```

Главное отличие между ними заключается в том, что имя массива heart является константой, а указатель head – переменной. Во что это выливается на практике?

Прежде всего, в обоих случаях можно применять форму записи с массивом:

```
for (i = 0; i < 7; i++)
   putchar(heart[i]);
putchar('\n');
for (i = 0; i < 7; i++)
   putchar(head[i]));
putchar('\n');
```

Получается следующий вывод:

```
Я люблю
Я люблю
```

Также в обоих случаях можно использовать добавление значения к указателю:

```
for (i = 0; i < 7; i++)
   putchar(*(heart + i));
putchar('\n');
for (i = 0; i < 7; i++)
   putchar(*(head + i));
putchar('\n');
```

И снова получается тот же самый вывод:

```
Я люблю
Я люблю
```

Однако операция инкремента может применяться только в версии с указателем:

```
while (*(head) != '\0')   /* остановиться в конце строки         */
   putchar(*(head++));    /* вывести символ, переместить указатель */
```

Этот код дает следующий вывод:

```
Я люблю Милли!
```

Предположим, вы хотите, чтобы head и heart совпадали. Тогда можно записать так:

```
head = heart;          /* head теперь указывает на массив heart */
```

В результате head будет указывать на первый элемент массива heart.

Однако следующий оператор не допускается:

```
heart = head;          /* недопустимая конструкция */
```

Ситуация аналогична случаю с операторами x = 3; и 3 = x;. В левой части оператора присваивания должна быть переменная, или в более общем смысле *l-значение*, такое как *p_int. Кстати, оператор head = heart; не приводит к затиранию строки "Я люблю Милли!"; она всего лишь меняет адрес, хранящийся в head. Однако если адрес строки "Я люблю Милли!" не будет сохранен где-то в другом месте, вы не сможете

получить доступ к этой строке после того, как head станет указывать на другую ячейку памяти.

Существует способ изменить сообщение heart — для этого нужно обращаться к отдельным элементам массива:

```
heart[8]= 'M';
```

или

```
*(heart + 8) = 'M';
```

Элементы массива являются переменными (если только массив не объявлен как const), но *имя* массива — это не переменная.

Давайте возвратимся к инициализации указателя, в которой модификатор const не используется:

```
char * word = "дело";
```

Можно ли применить указатель для изменения этой строки?

```
word[2] = 'п';  // допустимо??
```

Ваш компилятор может разрешить подобное, но согласно текущему стандарту C, поведение в этом случае не определено. Такой оператор может, например, привести к ошибке доступа в память. Причина связана с тем, что, как упоминалось ранее, компилятор может выбрать вариант представления всех идентичных строковых литералов в виде единственной копии в памяти. Например, все приведенные ниже операторы могут ссылаться на единственную ячейку памяти, в которой хранится строка "Клингон":

```
char * p1 = "Клингон";
p1[0] = 'Ф';    // все ли правильно?
printf("Клингон");
printf(": берегитесь %sцев!\n", "Клингон");
```

Это значит, что компилятор может заменить каждый экземпляр строкового литерала "Клингон" одним и тем же адресом. Если компилятор использует представление в виде единственной копии и разрешит произвести замену p1[0] на 'Ф', то это затронет все случаи использования данной строки, поэтому операторы, выводящие строковый литерал "Клингон", в действительности отобразят строку "Флингон":

```
Клингон: берегитесь Флингонцев!
```

На самом деле в прошлом некоторые компиляторы вели себя таким путающим образом, в то время как другие генерировали программы, которые в подобных случаях завершались аварийно. Поэтому рекомендуется при инициализации указателя строковым литералом применять модификатор const:

```
const char * p1 = "Клингон";  // рекомендуемое использование
```

Тем не менее, инициализация строковым литералом массива, отличного const, не влечет за собой проблем такого рода, поскольку массив получает копию исходной строки.

Короче говоря, не используйте указатель на строковый литерал, если вы планируете изменять строку.

Массивы символьных строк

Часто удобно иметь массив символьных строк. Тогда для доступа к разным строкам можно применять индекс. В листинге 11.4 продемонстрированы два подхода: массив указателей на строки и массив из массивов типа char.

Листинг 11.4. Программа arrchar.c

```
// arrchar.c - массив указателей, массив строк
#include <stdio.h>
#define SLEN 40
#define LIM 5
int main(void)
{
   const char *mytalents[LIM] = {
      "Мгновенное складывание чисел",
      "Точное умножение", "Накапливание данных",
      "Исполнение инструкций с точностью до буквы",
      "Знание языка программирования C"
   };
   char yourtalents[LIM][SLEN] = {
      "Хождение по прямой",
      "Здоровый сон", "Просмотр телепередач",
      "Рассылка писем", "Чтение электронной почты"
   };
   int i;

   puts("Сравним наши таланты.");
   printf ("%-52s %-25s\n", "Мои таланты", "Ваши таланты");
   for (i = 0; i < LIM; i++)
      printf("%-52s %-25s\n", mytalents[i], yourtalents[i]);
   printf("\nразмер mytalents: %zd, размер yourtalents: %zd\n",
          sizeof(mytalents), sizeof(yourtalents));

   return 0;
}
```

В результате получается следующий вывод:

```
Сравним наши таланты.
Мои таланты                                 Ваши таланты
Мгновенное складывание чисел                Хождение по прямой
Точное умножение                            Здоровый сон
Накапливание данных                         Просмотр телепередач
Исполнение инструкций с точностью до буквы  Рассылка писем
Знание языка программирования C             Чтение электронной почты

размер mytalents: 40, размер yourtalents: 200
```

Во многих отношениях массивы mytalents и yourtalents очень похожи. Каждый представляет по пять строк. Когда используется один индекс, как в mytalents[0] и yourtalents[0], результатом будет одиночная строка. Подобно тому, как значением mytalents[1][2] является 'ч', т.е. третий символ во второй строке, представленной массивом mytalents, yourtalents[1][2] — это 'ж', т.е. третий символ второй строки, представленной массивом yourtalents. Оба массива инициализируются в одинаковой манере.

Но имеются и различия. Массив mytalents — это массив из пяти указателей, занимающий в нашей системе 40 байтов. Но yourtalents — массив, состоящий из пяти массивов по 40 значений char и занимающий в нашей системе 200 байтов. Таким образом, тип массива mytalents отличается от типа yourtalents, несмотря на то, что и mytalents[0], и yourtalents[0] — это строки. Указатели в mytalents указывают на места размещения строковых литералов, применяемых для инициализации, которые хранятся в статической памяти. Однако массивы в yourtalents содержат копии строковых литералов, в результате чего каждая строка сохраняется дважды.

Более того, распределение памяти в массивах неэффективно, т.к. все элементы yourtalents должны иметь одинаковый размер, и этот размер должен быть достаточно большим, чтобы вместить самую длинную строку.

Один из способов восприятия этого различия – представление yourtalents в виде прямоугольного двумерного массива, все строки которого имеют одинаковую длину, в данном случае 40 байтов. В то же время mytalents можно представить в виде зубчатого массива с варьирующейся длиной строк. Эти два вида массивов показаны на рис. 11.2. (В действительности строки, на которые указывают элементы массива mytalents, не обязательно должны храниться последовательно в памяти, однако рисунок задуман в качестве иллюстрации различий в требованиях к хранению.)

Смысл всего сказанного в том, что при представлении набора строк, предназначенных для отображения, массив указателей более эффективен, чем массив символьных массивов. Однако существует и ограничение. Поскольку указатели в массиве mytalents указывают на строковые литералы, эти строки не должны изменяться. Тем не менее, содержимое массива yourtalents может изменяться. Поэтому, если предполагается изменение строк или требуется зарезервировать память для ввода строк, не следует использовать указатели на строковые литералы.

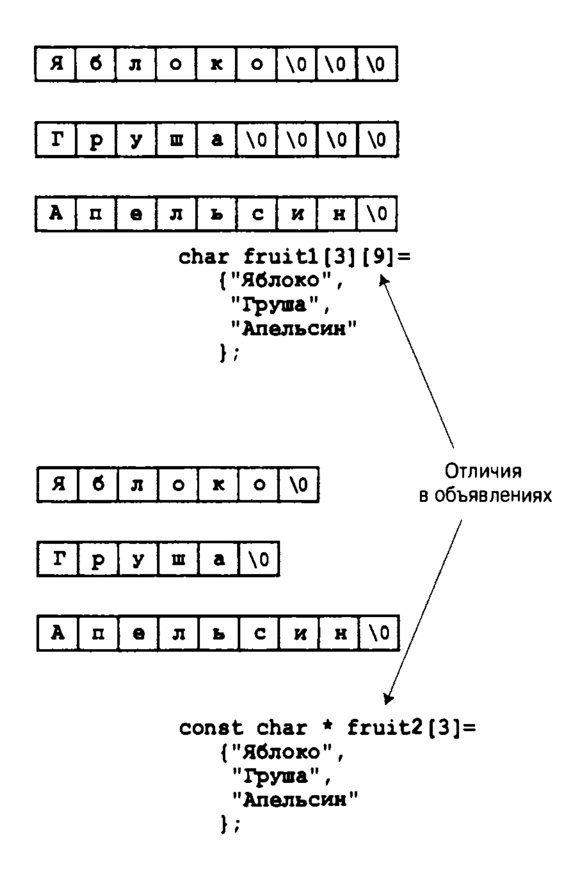


Рис. 11.2. Прямоугольные и зубчатые массивы

Указатели и строки

Возможно, вы уже заметили, что в этом обсуждении строк мы периодически ссылаемся на указатели. Большинство операций со строками в C фактически работают с указателями. В качестве примера рассмотрим короткую программу в листинге 11.5.

Листинг 11.5. Программа p_and_s.c

```
/* p_and_s.c -- указатели и строки */
#include <stdio.h>
int main(void)
{
   const char * mesg = "Не позволяйте себя запутать!";
   const char * copy;

   copy = mesg;
   printf("%s\n", copy);
   printf("mesg = %s; &mesg = %p; value = %p\n",
          mesg, &mesg, mesg);
   printf("copy = %s; &copy = %p; value = %p\n",
          copy, &copy, copy);

   return 0;
}
```

На заметку!

Если ваш компилятор не поддерживает спецификатор %p, замените его %u или %lu.

Глядя на эту программу, можно подумать, что она создает копию строки "Не позволяйте себя запутать!", и на первый взгляд вывод подтверждает это предположение:

```
Не позволяйте себя запутать!!
mesg = Не позволяйте себя запутать!; &mesg = 0x0012ff48; value = 0x0040a000
copy = Не позволяйте себя запутать!; &copy = 0x0012ff44; value = 0x0040a000
```

Но давайте взглянем на вывод printf() более внимательно. Прежде всего, значения mesg и copy выводятся в виде строки (%s). Здесь нет никакого сюрприза; все строки выглядят как "Не позволяйте себя запутать!".

Следующий элемент в каждой строке представляет адрес определенного указателя. Для конкретного запуска два указателя mesg и copy хранятся, соответственно, в ячейках 0x0012ff48 и 0x0012ff44.

Теперь обратите внимание на завершающий элемент по имени value. Это значение заданного указателя. Значением указателя является адрес, который он содержит. Как видите, mesg указывает на ячейку 0x0040a000, и то же самое можно сказать о copy. Следовательно, сама строка не копировалась. Оператор copy = mesg; всего лишь создает второй указатель, ссылающийся на ту же строку.

Тогда зачем все эти предосторожности? Почему бы просто не скопировать всю строку? Задайте себе вопрос: что эффективнее – копировать один адрес или, скажем, 50 отдельных элементов? Часто для решения задачи вполне достаточно только адреса. Если вам действительно необходима копия строки, т.е. ее дубликат, можете воспользоваться функцией strcpy() или strncpy(), как будет показано далее в главе.

Теперь, когда мы обсудили определение строк внутри программы, давайте переключимся на строки, вводимые с клавиатуры.

Ввод строк

Если в программе нужно ввести строку, понадобится сначала зарезервировать пространство для хранения этой строки и затем с помощью функции ввода извлечь строку.

Создание пространства под строку

Первое действие связано с подготовкой места, куда будет помещена строка после ее чтения. Как упоминалось ранее, это означает выделение пространства, достаточного для последующего чтения строк. Не следует ожидать, что компьютер подсчитает длину строки во время ее чтения и выделит необходимое пространство. Компьютер не будет делать это (если только вы не напишете специальную функцию). Например, предположим, что имеется такой код:

```
char *name;
scanf("%s", name);
```

Возможно, компилятор примет этот код, скорее всего, выдав предупреждение, но при чтении name строка может быть записана поверх данных или кода вашей программы и вызвать ее аварийное завершение. Причина в том, что функция scanf() копирует информацию по адресу, предоставленному аргументом, а в этом случае аргументом является неинициализированный указатель; name может указывать куда угодно. Большинство программистов находят такую ситуацию крайне забавной, но только не в собственных программах.

Проще всего решить эту проблему, включив в объявление явный размер массива:

```
char name[81];
```

Теперь name представляет собой адрес зарезервированного блока памяти размером 81 байт. Другая возможность предполагает применение функций выделения памяти из библиотеки C, и мы рассмотрим их в главе 12.

После установки пространства для строки ее можно прочитать. Библиотека C предлагает три функции, позволяющие считывать строки: scanf(), gets() и fgets(). Чаще других используется функция gets(), поэтому мы обсудим ее первой.

Неудачливая функция gets()

Вспомните, что при чтении строки функция scanf() и спецификатор %s обеспечивают считывание только одного слова. Однако часто желательно, чтобы программа могла читать сразу всю вводимую строку, а не одиночное слово. Именно такой цели на протяжении многих лет служила функция gets(). Это простая и легкая в использовании функция. Она читает всю строку вплоть до символа новой строки, отбрасывает этот символ и сохраняет остальные символы, добавляя нулевой символ, чтобы образовать строку C. Часто эта функция применяется в сочетании с функцией puts(), которая отображает строку с добавлением в конце символа новой строки. В листинге 11.6 приведен короткий пример.

Листинг 11.6. Программа getsputs.c

```
/* getsputs.c -- использование функций gets() и puts() */
#include <stdio.h>
#define STLEN 81
int main(void)
{
   char words[STLEN];
```

```
    puts("Введите строку.");
    gets(words);
    printf("Ваша строка, выведенная дважды:\n");
    printf("%s\n", words);
    puts(words);
    puts("Готово.");

    return 0;
}
```

Вот как мог бы выглядеть результат пробного запуска программы раньше:

```
Введите строку.
Я хотел бы ознакомиться с теорией работы со строками!
Ваша строка, выведенная дважды:
Я хотел бы ознакомиться с теорией работы со строками!
Я хотел бы ознакомиться с теорией работы со строками!
Готово.
```

Обратите внимание, что вся введенная строка кроме символа новой строки сохраняется в массиве `words`, а вызов `puts(words)` обеспечивает такой же эффект, как и вызов `printf("%s\n", words)`.

А вот так выглядит более новый вывод:

```
Введите строку.
warning: this program uses gets(), which is unsafe.
Предупреждение: эта программа использует функцию gets(), что небезопасно.
О, нет!
Ваша строка, выведенная дважды:
О, нет!
О, нет!
Готово.
```

Компилятор предпринял довольно необычное действие, вставив предупреждение в вывод программы! Таким образом, это сообщение будет отображаться при каждом запуске этой программы. Не все компиляторы будут поступать подобным образом. Некоторые из них могут выдавать предупреждение на этапе компиляции, но это не настолько привлекает внимание.

В чем же состоит проблема? Дело в том, что функция `gets()` не делает проверку, умещается ли вводимая строка в массив. Учитывая, что единственным аргументом функции является `words`, она просто не в состоянии выполнить такую проверку. Вспомните, что имя массива преобразуется в адрес его первого элемента. Следовательно, функции `gets()` известно только то, где начинается массив, но не то, сколько элементов в нем содержится.

Если строка ввода окажется слишком длинной, возникнет *переполнение буфера* — другими словами, символы переполнят предназначенное для них целевое пространство. Лишние символы могут просто попасть в неиспользуемую память и привести к проблемам, которые проявятся не сразу, или же они могут перезаписать другие данные в программе. И это — отнюдь не все возможные варианты. Ниже приведен пример запуска программы, в которой значение STLEN было переустановлено в 5, чтобы сделать переполнение буфера более вероятным.

```
Введите строку.
warning: this program uses gets(), which is unsafe.
Предупреждение: эта программа использует функцию gets(), что небезопасно.
Думаю, все будет отлично.
```

```
Ваша строка, выведенная дважды:
Думаю, все будет отлично.
Думаю, все будет отлично.
Готово.
Segmentation fault: 11
Ошибка сегментации: 11
```

"Ошибка сегментации" звучит не слишком здорово, правда? В системе Unix такое сообщение указывает на то, что программа попыталась получить доступ в память, которая не была для нее выделена.

К сожалению, язык С предоставляет много возможностей для того, чтобы неудачное программирование приводило к обескураживающим и трудно отслеживаемым ошибкам. Почему же тогда только недостаток функции `gets()` удостоен особого внимания? Вероятно потому, что ее непредсказуемое поведение создает риск для безопасности. В прошлом злоумышленники пользовались особенностями системных программ, в которых применялась функция `gets()`, для вставки и выполнения кода, который компрометировал безопасность системы.

На определенном этапе многие представители сообщества программистов на С рекомендовали исключить функцию `gets()` из программного словаря. Комитет, создавший стандарт С99, также опубликовал обоснование стандарта. В этом обосновании были признаны проблемы, связанные с использованием функции `gets()`, и применять ее не рекомендовалось. Вместе с тем в обосновании оправдывалось сохранение функции `gets()` в качестве части стандарта из-за ее удобства в случае корректного использования, а также из-за того, что она была частью огромного объема существующего кода.

Тем не менее, комитет по созданию стандарта C11 придерживался более строгих взглядов и исключил функцию `gets()` из стандарта. С другой стороны, стандарт определяет то, что компилятор должен поддерживать, а не то, что он не должен поддерживать. На практике большинство компиляторов продолжают поддерживать эту функцию ради обратной совместимости. Но, как это имеет место с применяемым нами компилятором, это отнюдь не идет ему на пользу.

Альтернативы функции gets()

Традиционной альтернативой `gets()` является функция `fgets()`, которая обладает несколько более сложным интерфейсом и немного по-другому обрабатывает вводимые данные. Кроме того, в стандарте C11 к общему набору добавлена функция `gets_s()`. Она больше похожа на `gets()` и ее легче использовать в существующем коде в качестве замены. Тем не менее, она является частью необязательного расширения семейства функций ввода-вывода `stdio.h`, поэтому компиляторы С стандарта C11 не обязаны ее поддерживать.

Функция fgets() (и fputs())

Функция `fgets()` предотвращает возможную проблему переполнения, принимая второй аргумент, который ограничивает количество считываемых символов. Эта функция предназначена для файлового ввода, что несколько затрудняет ее применение. Ниже перечислены ее отличия от функции `gets()`.

- Она принимает второй аргумент, задающий максимальное количество символов для чтения. Если этот аргумент имеет значение `n`, то функция `fgets()` прочитает `n-1` символов или будет читать до появления символа новой строки в зависимости от того, что произойдет раньше.

- Если функция fgets() сталкивается с символом новой строки, она сохраняет его в строке, в отличие от функции gets(), которая отбрасывает его.
- Функция fgets() принимает третий аргумент, указывающий файл, из которого должно производиться чтение. Для чтения с клавиатуры в качестве этого аргумента используется stdin (от *standard input* — стандартный ввод); этот идентификатор определен в stdio.h.

Поскольку функция fgets() обрабатывает символ новой строки как часть строки (при условии, что строка ввода имеет соответствующую длину), ее часто применяют совместно с функцией fputs(), которая работает подобно puts(), но не добавляет автоматически символ новой строки. Функция fputs() принимает второй аргумент, указывающий на файл, в который должна производиться запись. Для вывода на дисплей можно использовать аргумент stdout (от *standard output* — стандартный вывод).

В листинге 11.7 иллюстрируется поведение функций fgets() и fputs().

Листинг 11.7. Программа fgets1.c

```
/* fgets1.c -- использование функций fgets() и fputs() */
#include <stdio.h>
#define STLEN 15
int main(void)
{
   char words[STLEN];
   puts("Введите строку.");
   fgets(words, STLEN, stdin);
   printf("Ваша строка, выведенная дважды (с помощью puts(), а затем fputs()):\n");
   puts(words);
   fputs(words, stdout);
   puts("Введите еще одну строку.");
   fgets(words, STLEN, stdin);
   printf("Ваша строка, выведенная дважды (с помощью puts(), а затем fputs()):\n");
   puts(words);
   fputs(words, stdout);
   puts("Готово.");

   return 0;
}
```

Ниже показаны результаты пробного запуска:

```
Введите строку.
шарлотка
Ваша строка, выведенная дважды (с помощью puts(), а затем fputs()):
шарлотка

шарлотка
Введите еще одну строку.
клубничное песочное печенье
Ваша строка, выведенная дважды (с помощью puts(), а затем fputs()):
клубничное пес
клубничное песГотово.
```

Первая строка ввода, шарлотка, является достаточно короткой, чтобы функция fgets() прочитала ее и сохранила шарлотка\n\0 в массиве. Поэтому, когда функция puts() отображает строку и добавляет в вывод собственный символ новой строки, она порождает пустую строку вывода после строки шарлотка. Так как fputs() не добавляет символа новой строки, она не создает пустую строку.

Длина второй строки ввода, клубничное песочное печенье, превышает лимит на размер, поэтому fgets() считывает первые 14 символов и сохраняет в массиве строку клубничное пес\0. И снова функция puts() добавляет символ новой строки в вывод, а fputs() не делает этого.

Функция fgets() возвращает указатель на char. Если все проходит нормально, она просто возвращает тот же адрес, который был ей передан в первом аргументе. Однако если функция встречает конец файла, она возвращает специальный указатель, называемый *нулевым*. Такой указатель гарантированно не ссылается на реальные данные, поэтому может применяться для отражения особого случая. В коде он может быть представлен цифрой 0 или, что более распространено в C, макросом NULL. (Функция возвращает NULL также в ситуации, когда произошла какая-то ошибка чтения.) В листинге 11.8 показан простой цикл, который читает и повторяет текст до тех пор, пока функция fgets() не встретит конец файла или не выполнит считывание пустой строки — т.е. строки, первым символом которой является символ новой строки.

Листинг 11.8. Программа fgets2.c

```
/* fgets2.c -- использование функций fgets() и fputs() */
#include <stdio.h>
#define STLEN 10
int main(void)
{
   char words[STLEN];

   puts("Введите строки (или пустую строку для выхода из программы):");
   while (fgets(words, STLEN, stdin) != NULL && words[0] != '\n')
      fputs(words, stdout);
   puts("Готово.");

   return 0;
}
```

Вот как выглядит вывод из этой программы:

```
Введите строки (или пустую строку для выхода из программы):
Кстати говоря, функция gets()
Кстати говоря, функция gets()
также возвращает пустой указатель, если она
также возвращает пустой указатель, если она
встречает признак конца файла.
встречает признак конца файла.

Готово.
```

Интересно отметить, что хотя значение STLEN равно 10, похоже, программа не испытывает проблем при обработке строк ввода, длина которых значительно превышает этот предел. Дело в том, что в данной программе функция fgets() читает STLEN-1 (т.е. 9) символов за раз. Поэтому она начинает с чтения строки "Кстати го", сохраняя ее как Кстати го\0. Затем fputs() отображает эту строку, но при этом не переходит на следующую строку вывода. Далее функция fgets() возобновляет чтение с того места, где она остановилась, и считывает "воря, фун", сохранив ее как воря, фун\0. Функция fputs() отображает эту строку в той же строке, в которой она находилась ранее. Затем fgets() возобновляет чтение ввода продолжая до тех пор, пока не останется прочитать только "()\n"; функция fgets() сохраняет строку ()\n\0, функция fputs() отображает ее, а внутренний символ новой строки приводит к перемещению курсора на следующую строку.

В системе используется буферизированный ввод-вывод. Это означает, что введенные данные сохраняются во временной памяти (буфере) до тех пор, пока не будет нажата клавиша <Enter>. В результате этого к введенным данным добавляется символ новой строки, и вся строка передается функции `fgets()`. При выводе функция `fputs()` передает символы в другой буфер, и после отправки символа новой строки содержимое буфера передается дисплею.

Тот факт, что функция `fgets()` сохраняет символ новой строки, порождает проблему, но и предоставляет дополнительную возможность. Проблема заключается в том, что сохранение символа новой строки в виде части строки может быть нежелательным. Преимущество же в том, что присутствие или отсутствие символа новой строки в сохраненной строке может служить индикатором того, была ли прочитана вся строка. Если этого символа нет, можно принимать решение о том, что делать с оставшейся частью строки.

Во-первых, как избавиться от символа новой строки? Один из способов — его поиск в сохраненной строке и замена нулевым символом:

```
while (words[i] != '\n')  // предполагается, что \n присутствует в words
    i++;
words[i] = '\0';
```

Во-вторых, как быть, если в строке ввода по-прежнему остаются символы? Разумный подход на случай, если вся строка не помещается в целевом массиве, предусматривает игнорирование не умещающейся части:

```
while (getchar() != '\n') // чтение без сохранения
    continue;             // ввода, включая \n
```

В листинге 11.9 к этим базовым идеям добавляется дополнительная проверка, что дает в итоге код, который читает строки ввода, удаляет сохраненные символы новой строки, если таковые имеются, и отбрасывает часть строки, которая не умещается в выделенную область памяти.

Листинг 11.9. Программа `fgets3.c`

```
/* fgets3.c -- использование функции fgets() */
#include <stdio.h>
#define STLEN 10
int main(void)
{
    char words[STLEN];
    int i;
    puts("Введите строки (или пустую строку для выхода из программы):");
    while (fgets(words, STLEN, stdin) != NULL && words[0] != '\n')
    {
        i = 0;
        while (words[i] != '\n' && words[i] != '\0')
            i++;
        if (words[i] == '\n')
            words[i] = '\0';
        else  // требуется наличие words[i] == '\0'
            while (getchar() != '\n')
                continue;
        puts(words);
    }
    puts("Готово.");
    return 0;
}
```

Цикл

```
while (words[i] != '\n' && words[i] != '\0')
    i++;
```

выполняет проход по строке до тех пор, пока не встретит символ новой строки или нулевой символ (в зависимости от того, что произойдет раньше). Если найденный символ является символом новой строки, то следующий за циклом оператор if заменяет его нулевым символом. В противном случае часть else отбрасывает остаток строки ввода. Ниже показаны результаты пробного запуска:

```
Введите строки (или пустую строку для выхода из программы):
Эта
Эта
программа, похоже,
программа
не желает принимать длинные строки.
не желает
Но, тем не менее, она не стопорится на
Но, тем н
длинных строках.
длинных с

Готово.
```

Нулевой символ и нулевой указатель

В листинге 11.9 присутствуют и нулевой символ, и нулевой указатель. Концептуально эти две "нулевых" сущности отличаются друг от друга. Нулевой символ, или \0, является символом, применяемым для пометки конца строки С. Этот символ имеет код, равный нулю. Поскольку данный код не соответствует никакому другому символу, нулевой символ не может случайно появиться в какой-то другой части строки.

Нулевой указатель, или NULL, имеет значение, которое не соответствует какому-то допустимому адресу данных. Он часто используется в функциях, в противном случае возвращающих допустимые адреса, для указания на некоторую специальную ситуацию, такую как обнаружение признака конца файла или невозможность выполнения действия ожидаемым образом.

Итак, нулевой символ имеет целочисленный тип, а нулевой указатель — тип указателя. Иногда путаница возникает из-за того, что числовым представлением их обоих может оказаться значение 0. Но концептуально — это различные типы 0. Кроме того, нулевой символ, будучи символом, занимает один байт, тогда как нулевой указатель, как адрес, обычно занимает четыре байта.

Функция gets_s()

Необязательная функция gets_s() в C11, подобно fgets(), применяет аргумент для ограничения количества читаемых символов. С учетом определений из листинга 11.9 следующий код будет считывать строку ввода в массив words, обеспечивая появление символа новой строки в числе первых 9 символов ввода:

```
gets_s(words, STLEN);
```

Ниже описаны три основных отличия этой функции от fgets().

- Функция gets_s() просто выполняет чтение из стандартного ввода, поэтому она не нуждается в третьем аргументе.
- Если функция gets_s() считывает символ новой строки, то отбрасывает его, а не сохраняет.

- Если gets_s() прочитает максимальное количество символов, и среди них символ новой строки отсутствует, она предпринимает несколько действий. Функция устанавливает первый символ целевого массива в нулевой символ. Затем она читает и отбрасывает последующие введенные данные, пока не встретится символ новой строки или признак конца файла. Наконец, функция возвращает нулевой указатель. Она вызывает зависящую от реализации функцию "обработчика" (или же выбранную вами функцию), которая может привести к выходу из программы или прекращению ее работы.

Второе отличие означает, что до тех пор, пока строка ввода не слишком длинная, функция gets_s() ведет себя подобно gets(), благодаря чему gets() проще заменить функцией gets_s(), чем fgets(). Третье отличие означает необходимость в обучении применению этой функции.

Давайте сравним приемлемость функций gets(), fgets() и gets_s(). Если строка ввода умещается в целевую область памяти, то все три функции работают успешно. Но функция fgets() включает в строку символ новой строки, поэтому может возникнуть необходимость в коде для его замены нулевым символом.

А что происходит, если длина строки ввода превышает заданный размер? Тогда использование функции gets() будет небезопасным — могут быть повреждены данные и нарушена безопасность. Функция gets_s() безопасна, но если вы не хотите, чтобы программа прекратила работу или выполнила выход каким-нибудь способом, придется подумать над тем, как написать и зарегистрировать специальные "обработчики". Кроме того, если все же удается сохранить программу в работоспособном состоянии, функция gets_s() отбросит остальную часть строки ввода независимо от вашего желания. В случае, когда строка не умещается в заданную область памяти, с функцией fgets() иметь дело проще, чем с двумя другими функциями, и она предоставляет больше возможных вариантов. Если нужно, чтобы программа обработала остальную часть строки ввода, это можно сделать, как показано в листинге 11.8. Если же требуется, чтобы программа обработала остальную часть строки ввода, это также возможно, как видно в листинге 11.9.

Таким образом, когда ввод не соответствует ожиданиям, функция gets_s() является менее удобной и гибкой по сравнению с fgets(). Вероятно, это одна из причин, по которой функция gets_s() представляет собой лишь необязательное расширение библиотеки С. И с учетом необязательности gets_s() обычно лучшим выбором будет функция fgets().

Функция s_gets()

В листинге 11.9 представлен один из способов применения функции fgets(): чтение всей строки и замена символа новой строки нулевым символом либо чтение части строки, умещающейся в заданную область памяти, и отбрасывание остальных символов — т.е. своего рода разновидность функции gets_s(), но без дополнительных препятствий. Ни одна из стандартных функций не удовлетворяет этому описанию, но мы можем создать такую функцию самостоятельно. Она пригодится в будущих примерах. В листинге 11.10 демонстрируется один из подходов.

Листинг 11.10. Функция s_gets()

```
char * s_gets(char * st, int n)
{
   char * ret_val;
   int i = 0;
```

```
   ret_val = fgets(st, n, stdin);
   if (ret_val)  // т.е. ret_val != NULL
   {
      while (st[i] != '\n' && st[i] != '\0')
         i++;
      if (st[i] == '\n')
         st[i] = '\0';
      else  // требуется наличие words[i] == '\0'
         while (getchar() != '\n')
            continue;
   }
   return ret_val;
}
```

В то время как функция `fgets()` возвращает `NULL`, указывая на конец файла или ошибку чтения, функция `s_gets()` пропускает обработку остальных данных. В противном случае она имитирует работу программы из листинга 11.9, заменяя в строке символ новой строки нулевым символом, если названный символ присутствует, и отбрасывая остальную часть строки в противоположной ситуации. Затем она возвращает то же самое значение, что и `fgets()`. Мы будем использовать эту функцию в последующих примерах.

Возможно, вас интересует, в чем смысл отбрасывания остальной части слишком длинной строки. Проблема в том, что если остальную часть строки оставить, она становится входными данными для следующего оператора чтения. Это может, например, вызвать аварийное завершение программы, если следующий оператор чтения ожидает значение типа `double`. Отбрасывание остальной части строки поддерживает операторы чтения в состоянии синхронизации с вводом с клавиатуры.

Наша функция `s_gets()` не идеальна. Наиболее серьезный ее недостаток в том, что она молча отбрасывает лишние входных данных, не ставя об этом в известность ни программу, ни пользователя, и тем самым лишая его других возможностей, таких как повторение ввода или нахождение большего объема памяти. Еще один недостаток — отсутствие мер на случай неправильного применения вроде передачи размера, равного 1 или меньше. Тем не менее, эта функция вполне может служить заменой функции `gets()` в наших примерах.

Функция scanf()

Давайте снова возвратимся к функции `scanf()`. Ранее для чтения строки мы использовали функцию `scanf()` с форматом `%s`. Основное отличие между функциями `scanf()` и `fgets()` связано с тем, как они определяют момент достижения конца строки: функция `scanf()` больше ориентирована на "получение слова", а не на "получение строки". Функция `gets()`, как вы уже видели, принимает все символы вплоть до первого символа новой строки, как это делает `fgets()`, если строка является достаточно короткой.

Функция `scanf()` располагает двумя возможностями для прекращения ввода. При любом варианте строка начинается с первого встреченного непробельного символа. Если задан формат `%s`, строка продолжается до следующего (не включая его) пробельного символа (символа пробела, табуляции или новой строки). Если указана ширина поля, как в `%10s`, функция `scanf()` читает до получения 10 символов или до появления первого пробельного символа, в зависимости от того, что произойдет раньше (рис. 11.3).

Оператор ввода	Исходная очередь ввода *	Содержимое строки имени	Остальная часть очереди
scanf("%s", name);	Иванов□Иван	Иванов	□Иван
scanf("%5s", name);	Иванов□Иван	Ивано	в□Иван
scanf("%5s", name);	Иван□Иванов	Иван	□Иванов

* □ представляет пробельный символ.

Рис. 11.3. Ширина поля и функция scanf ()

Вспомните, что функция scanf() возвращает целочисленное значение, равное количеству успешно прочитанных элементов, или EOF при обнаружении конца файла.

В листинге 11.11 иллюстрируется работа функции scanf(), когда задана ширина поля.

Листинг 11.11. Программа scan_str.c

```
/* scan_str.c -- использование функции scanf() */
#include <stdio.h>
int main(void)
{
   char name1[11], name2[11];
   int count;
   printf("Введите два имени.\n");
   count = scanf("%5s %10s", name1, name2);
   printf("Прочитано %d имени: %s и %s.\n",
         count, name1, name2);
   return 0;
}
```

Вот результаты трех пробных запусков программы:

```
Введите два имени.
Джейн Покер
Прочитано 2 имени: Джейн и Покер.
Введите два имени.
Иван Миттельшпиль
Прочитано 2 имени: Иван и Миттельшпи.
Введите два имени.
Вениамин Каверин
Прочитано 2 имени: Вениа и мин.
```

В первом запуске оба имени вписываются в разрешенные предельные размеры. Во втором запуске были прочитаны только первые 10 символов имени Миттельшпиль, поскольку мы применяем формат %10s. В третьем запуске в name2 попадают последние три буквы из имени Вениамин, т.к. второй ввод возобновляется там, где закончился первый ввод: в данном случае — внутри слова Вениамин.

В зависимости от природы желаемого ввода, для чтения текста с клавиатуры может быть удобнее пользоваться функцией fgets(). Например, функция scanf() не особенно подходит для ввода названия книги или песни, если только название не состоит из одного слова. Типичным применением scanf() является чтение и преобразование смеси данных разных типов в некоторой стандартной форме. Например, если входная строка содержит название инструмента, складской номер и цену за штуку, то можно воспользоваться scanf() либо написать собственную функцию, которая будет

выполнять проверку на предмет ошибок ввода. Если вы хотите обрабатывать ввод по слову за раз, можете применять scanf().

Функция scanf() страдает тем же потенциальным недостатком, что и gets(): она может приводить к переполнению, если вводимое слово не умещается в целевую область памяти. Но для предотвращения такого переполнения можно использовать параметр ширины поля в спецификаторе %s.

Вывод строк

Теперь давайте перейдем от ввода строк к их выводу. Мы снова будем применять библиотечные функции. Для вывода строк в С доступны три стандартных библиотечных функции: puts(), fputs() и printf().

Функция puts()

Функция puts() очень проста в использовании. Ей нужно только передать в качестве аргумента адрес строки. В листинге 11.12 демонстрируется несколько способов из множества доступных.

Листинг 11.12. Программа put_out.c

```
/* put_out.c -- использование функции puts() */
#include <stdio.h>
#define DEF "Я - строка, определенная директивой #define."
int main(void)
{
    char str1[80] = "Массив был инициализирован моим значением.";
    const char * str2 = "Указатель был инициализирован моим значением.";
    puts("Я - аргумент функции puts().");
    puts(DEF);
    puts(str1);
    puts(str2);
    puts(&str1[5]);
    puts(str2+4);
    return 0;
}
```

Вывод имеет следующий вид:

```
Я - аргумент функции puts().
Я - строка, определенная директивой #define.
Массив был инициализирован моим значением.
Указатель был инициализирован моим значением.
в был инициализирован моим значением.
атель был инициализирован моим значением.
```

Как и в предшествующих примерах, каждая строка выводится в собственной строке, потому что при отображении строки функция puts() автоматически добавляет символ новой строки.

Этот пример напоминает, что фразы, заключенные в двойные кавычки, представляют собой строковые константы и трактуются как адреса. Кроме того, имена строковых массивов также считаются адресами. Выражение &str1[5] — это адрес шестого элемента в массиве str1. Этот элемент содержит символ 'в', и именно он puts() применяет в качестве начальной точки. Аналогично, str2+4 указывает на ячейку памяти, содержащую второй символ 'а' из строки "Указатель", поэтому вывод начинается с него.

Как функция puts() узнает, когда остановиться? Она прекращает вывод, когда встречает нулевой символ, так что лучше, чтобы он был в строке. Не повторяйте ошибку, проиллюстрированную в листинге 11.13!

Листинг 11.13. Программа nono.c

```
/* Программа nono.c -- не делайте так! */
#include <stdio.h>
int main(void)
{
   char side_a[] = "Сторона А";
   char dont[] = {'У', 'Р', 'А', '!' };
   char side_b[] = "Сторона Б";

   puts(dont);   /* dont не является строкой */

   return 0;
}
```

Поскольку в массиве dont отсутствует завершающий нулевой символ, он не является строкой, поэтому функция puts() не знает, где остановиться. Она будет выводить содержимое ячеек памяти, следующих за dont, пока не обнаружит нулевой символ где-то в другом месте. Чтобы гарантировать, что нулевой символ не окажется слишком далеко, массив dont в программе хранится между двумя настоящими строками. Вот как выглядит вывод:

```
УРА!Сторона А
```

Использованный здесь компилятор разместил в памяти массив side_a после массива dont, так что функция puts() продолжает движение до тех пор, пока не наталкивается на нулевой символ в массиве side_a. В зависимости от того, как ваш компилятор размещает данные в памяти, вы можете получить другие результаты. Что будет, если из программы убрать массивы side_a и side_b? Обычно в памяти содержится много нулей, и при некоторой доле везения функция puts() сможет найти один из них довольно быстро, однако не стоит на это полагаться.

Функция fputs()

Функция fputs() представляет собой версию puts(), ориентированную на файлы. Важные отличия между ними описаны ниже.

- Функция fputs() принимает второй аргумент, указывающий файл, в который должна производиться запись. Для вывода на дисплей можно применять аргумент stdout (от *standard output* – стандартный вывод), который определен в stdio.h.
- В отличие от puts(), функция fputs() не делает автоматическое дополнение вывода символом новой строки.

Обратите внимание, что gets() отбрасывает символ новой строки из введенных данных, но puts() добавляет его в вывод. С другой стороны, fgets() сохраняет символ новой строки во введенных данных, а fputs() не помещает его в вывод. Предположим, что вы хотите реализовать цикл, в котором строка читается и выводится в следующей строке на экране. Вы можете поступить так, как показано далее:

```
char line[81];
while (gets(line))  // эквивалентно while (gets(line) != NULL)
   puts(line);
```

Вспомните, что функция gets() возвращает нулевой указатель, если обнаруживает конец файла. Нулевой указатель интерпретируется как ноль, или ложное значение, поэтому цикл прекращается. Либо можно поступить так:

```
char line[81];
while (fgets(line, 81, stdin))
    fputs(line, stdout);
```

В первом цикле строка из массива line отображается в собственной строке на экране, поскольку puts() добавляет символ новой строки. Во втором цикле строка из массива line отображается в собственной строке на экране из-за того, что fgets() сохраняет символ новой строки.

Следует отметить, что если вы смешиваете ввод fgets() с выводом puts(), то получите по два символа новой строки для каждой строки на экране. Важно понимать, что функция puts() спроектирована для работы с gets(), а функция fputs() — для работы с fgets().

Разумеется, мы упомянули о функции gets() только для того, чтобы вы знали, как она работает, если вы встретите ее в коде, и вовсе не призываем пользоваться ею.

Функция printf()

Мы довольно подробно обсуждали функцию printf() в главе 4. Как и puts(), она принимает в качестве аргумента адрес строки. Функция printf() менее удобна в употреблении, чем puts(), но она более универсальна, т.к. способна форматировать различные типы данных.

Одно из отличий заключается в том, что функция printf() не выводит автоматически каждую строку в новой строке на экране. Вместо этого вы должны самостоятельно указывать, где должны начинаться новые строки. Таким образом,

```
printf("%s\n", string);
```

приводит к тому же результату, что и

```
puts(string);
```

Как видите, первая форма длиннее. Она также требует большего времени на выполнение (правда, не настолько, чтобы это стало заметным). С другой стороны, printf() упрощает объединение нескольких строк в одной строке вывода. Например, следующий оператор объединяет в одну строку вывода слово Хорошо, имя пользователя и символьную строку, определенную с помощью #define:

```
printf("Хорошо, %s, %s\n", name, MSG);
```

Возможность самостоятельного создания функций

При вводе и выводе вы не ограничены только функциями стандартной библиотеки C. Если они недоступны или по какой-то причине не нравятся, можете подготовить собственные версии на основе функций getchar() и putchar(). Предположим, что вам нужна функция, подобная puts(), которая не добавляет автоматически символ новой строки. В листинге 11.14 продемонстрирован один из способов создания такой функции.

Листинг 11.14. Функция put1()

```
/* put1.c -- выводит строку без добавления символа \n */
#include <stdio.h>
void put1(const char * string)  /* строка не изменяется */
{
   while (*string != '\0')
      putchar(*string++);
}
```

Указатель string на char изначально ссылается на первый элемент переданного аргумента. Поскольку эта функция не изменяет строку, применяется модификатор const. После того, как содержимое этого элемента выведено, указатель инкрементируется и указывает на следующий элемент. Это продолжается до тех пор, пока указатель не будет ссылаться на элемент, содержащий нулевой символ. Вспомните, что операция ++ имеет более высокий приоритет, чем *, так что вызов putchar(*string++) выводит значение, на которое указывает string, но инкрементирует сам указатель string, а не символ, на который он ссылается.

Функцию put1.c можно рассматривать как модель для написания функций обработки строк. Поскольку каждая строка содержит нулевой символ, обозначающий ее конец, передавать функции размер строки не нужно. Вместо этого функция обрабатывает символы по очереди, пока не встретит нулевой символ.

Несколько более длинный код функции предусматривает использование формы записи с массивом:

```
int i = 0;
while (string[i]!= '\0')
   putchar(string[i++]);
```

Здесь требуется дополнительная переменная для индекса.

Многие программисты на С будут применять следующую проверку для цикла while:

```
while (*string)
```

Когда string указывает на нулевой символ, *string имеет значение 0, что прекращает цикл. Такой подход определенно требует меньшего набора с клавиатуры, чем предыдущая версия. Тем, кто не знаком с практикой программирования на С, этот прием менее очевиден. Однако данный подход получил широкое распространение, и ожидается, что программисты на С должны его знать.

На заметку!

Почему в листинге 11.14 в качестве формального аргумента используется const char * string, а не const char string[]? Формально они эквивалентны, поэтому будет работать любая форма. Одна из причин применения формы записи с квадратными скобками — желание напомнить пользователю, что данная функция обрабатывает массив. Тем не менее, в случае строк фактическим аргументом может быть имя массива, строка в кавычках или переменная, которая была объявлена с типом char*. Использование const char * string напоминает о том, что фактическим аргументом не обязательно должен быть массив.

Предположим, что вам необходима функция, похожая на puts(), которая также сообщает, сколько символов было выведено. Как демонстрируется в листинге 11.15, добавить такую возможность легко.

Листинг 11.15. Функция put2()

```
/* put2.c -- выводит строку и подсчитывает символы */
#include <stdio.h>
int put2(const char * string)
{
   int count = 0;
   while (*string)        /* общепринятый подход */
   {
      putchar(*string++);
      count++;
   }
   putchar('\n');         /* символ новой строки не учитывается */
   return(count);
}
```

Следующий вызов функции выводит строку пицца:

```
put1("пицца");
```

Показанный ниже вызов возвращает также количество символов, присвоенных переменной num (в данном случае 5):

```
num = put2("пицца");
```

В листинге 11.16 представлен драйвер, предназначенный для тестирования put1() и put2(), а также вложенных вызовов этих функций.

Листинг 11.16. Программа put_put.c

```
// put_put.c -- функции вывода, определенные пользователем
#include <stdio.h>
void put1(const char *);
int put2(const char *);
int main(void)
{
   put1("Если бы у меня было столько денег,");
   put1(" сколько я мог бы потратить,\n");
   printf("Получилось %d символов.\n",
         put2("то я никогда не заботился бы о починке старых башмаков."));
   return 0;
}
void put1(const char * string)
{
   while (*string)  /* эквивалентно *string != '\0' */
      putchar(*string++);
}
int put2(const char * string)
{
   int count = 0;
   while (*string)
   {
      putchar(*string++);
      count++;
   }
   putchar('\n');
   return(count);
}
```

Мы применяем printf() для вывода значения функции put2(), но в процессе выяснения этого значения эта функция сначала должна быть выполнена, что приводит к выводу строки. Ниже показан вывод:

```
Если бы у меня было столько денег, сколько я мог бы потратить,
то я никогда не заботился бы о починке старых башмаков.
Получилось 55 символов.
```

Строковые функции

Библиотека C предоставляет несколько функций обработки строк; в ANSI C прототипы этих функций содержатся в заголовочном файле string.h. Мы ознакомимся с наиболее полезными и распространенными функциями: strlen(), strcat(), strncat(), strcmp(), strncmp(), strcpy() и strncpy(). Мы также исследуем функцию sprintf(), которая поддерживается заголовочным файлом stdio.h. Полный перечень семейства функций string.h приведен в разделе V приложения Б.

Функция strlen()

Как вы уже знаете, функция strlen() находит длину строки. Она используется в следующем примере функции, которая сокращает длинные строки:

```
void fit(char *string, unsigned int size)
{
   if (strlen(string) > size)
      string[size] = '\0';
}
```

Функция изменяет строку, поэтому в ее заголовке при объявлении формального параметра string модификатор const не указан.

Функцию fit() можно протестировать с помощью программы, приведенной в листинге 11.17. Обратите внимание, что в коде применяется конкатенация строковых литералов C.

Листинг 11.17. Программа test_fit.c

```
/* test_fit.c -- использование функции укорачивания строк */
#include <stdio.h>
#include <string.h>  /* содержит прототипы строковых функций */
void fit(char *, unsigned int);

int main(void)
{
   char mesg[] = "Все должно быть максимально простым,"
   " но не более.";

   puts(mesg);
   fit(mesg, 35);
   puts(mesg);
   puts("Рассмотрим еще несколько строк.");
   puts(mesg + 36);
   return 0;
}
void fit(char *string, unsigned int size)
{
   if (strlen(string) > size)
      string[size] = '\0';
}
```

Вывод этой программы имеет следующий вид:

```
Все должно быть максимально простым, но не более.
Все должно быть максимально простым
Рассмотрим еще несколько строк.
 но не более.
```

Функция `fit()` помещает символ `'\0'` в 36-й элемент массива вместо символа запятой. Она останавливается при обнаружении первого нулевого символа, игнорируя оставшуюся часть массива. Тем не менее, остальная часть массива никуда не делась, как показывает следующий вызов функции:

```
puts(mesg + 36);
```

Выражение `mesg + 36` дает адрес элемента `mesq[36]`, которым является символ пробела. Таким образом, `puts()` отображает этот символ и продолжает работу до тех пор, пока не столкнется с исходным нулевым символом. На рис. 11.4 показано, что происходит в этой программе (на примере более короткой строки).

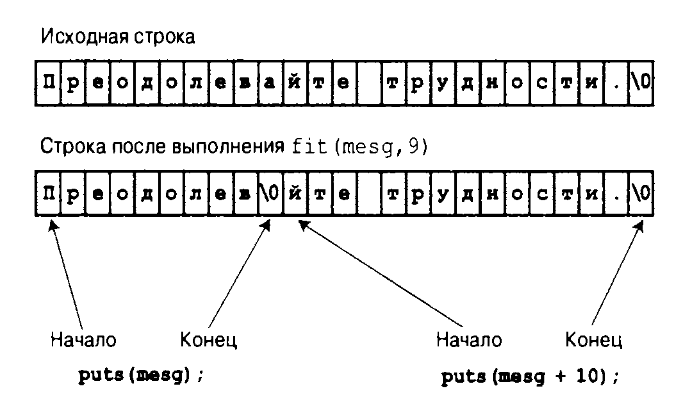


Рис. 11.4. Функция `puts()` *и нулевой символ*

Заголовочный файл `string.h` содержит прототипы для семейства строковых функций C, поэтому он и был включен в данном примере.

На заметку!

В некоторых системах, существовавших до выхода стандарта ANSI, вместо `string.h` используется заголовочный файл `strings.h`, а в ряде систем такой заголовочный файл вообще отсутствует.

Функция strcat()

Функция `strcat()` (от *string concatenation* – конкатенация строк) в качестве аргументов принимает две строки. Копия второй строки присоединяется в конец первой строки, и такая объединенная версия становится новой первой строкой. Вторая строка не изменяется. Функция `strcat()` имеет тип `char *` (т.е. указатель на `char`). Она возвращает значение своего первого аргумента – адрес первого символа строки, в конец которой была добавлена вторая строка.

В листинге 11.18 демонстрируются возможности функции `strcat()`. В коде также применяется функция `s_gets()`, которая была определена в листинге 11.10; вспомните, что `s_gets()` использует `fgets()` для чтения строки, а затем удаляет из нее символ новой строки, если он присутствует.

Листинг 11.18. Программа str_cat.c

```
/* str_cat.c -- объединяет две строки */
#include <stdio.h>
#include <string.h>  /* объявление strcat() */
#define SIZE 80
char * s_gets(char * st, int n);
int main(void)
{
   char flower[SIZE];
   char addon[] = " пахнет как старые валенки.";

   puts("Какой у вас любимый цветок?");
   if (s_gets(flower, SIZE))
   {
      strcat(flower, addon);
      puts(flower);
      puts(addon);
   }
   else
      puts("Обнаружен конец файла!");
   puts("Программа завершена.");

   return 0;
}

char * s_gets(char * st, int n)
{
   char * ret_val;
   int i = 0;

   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      while (st[i] != '\n' && st[i] != '\0')
         i++;
      if (st[i] == '\n')
         st[i] = '\0';
      else  // требуется наличие words[i] == '\0'
         while (getchar() != '\n')
            continue;
   }
   return ret_val;
}
```

Получаем следующие выходные данные:

```
Какой у вас любимый цветок?
Анемон
Анемон пахнет как старые валенки.
 пахнет как старые валенки.
Программа завершена.
```

В выводе видно, что строка `flower` изменилась, а строка `addon` – нет.

Функция strncat()

Функция `strcat()` не проверяет, умещается ли вторая строка в первый массив. Если вы не выделите достаточного пространства для первого массива, то столкнетесь с проблемой переполнения соседних ячеек памяти избыточными символами.

Конечно, можно заранее проверить длину с помощью strlen(), как показано в листинге 11.15. Обратите внимание, что эта функция добавляет 1 к общей длине, резервируя место для нулевого символа. В качестве альтернативы можно воспользоваться функцией strncat(), которая принимает второй аргумент, указывающий максимальное количество добавляемых символов. Например, вызов strncat(bugs, addon, 13) добавляет содержимое строки addon к bugs, останавливаясь после прохода 13 дополнительных символов или при обнаружении нулевого символа, в зависимости от того, что случится раньше. Следовательно, учитывая нулевой символ (который добавляется в любом случае), массив bugs должен иметь размер, достаточный для хранения исходной строки (без нулевого символа), максимум 13 дополнительных символов и завершающего нулевого символа. В листинге 11.19 эта информация применяется для вычисления значения переменной available, которая служит максимальным разрешенным количеством дополнительных символов.

Листинг 11.19. Программа join_chk.c

```
/* join_chk.c -- объединяет две строки, предварительно проверив размер */
#include <stdio.h>
#include <string.h>
#define SIZE 30
#define BUGSIZE 13
char * s_gets(char * st, int n);
int main(void)
{
   char flower[SIZE];
   char addon[] = " пахнет как старые валенки.";
   char bug[BUGSIZE];
   int available;
   puts("Какой у вас любимый цветок?");
   s_gets(flower, SIZE);
   if ((strlen(addon) + strlen(flower) + 1) <= SIZE)
      strcat(flower, addon);
   puts(flower);
   puts("Какое у вас любимое насекомое?");
   s_gets(bug, BUGSIZE);
   available = BUGSIZE - strlen(bug) - 1;
   strncat(bug, addon, available);
   puts(bug);
   return 0;
}
char * s_gets(char * st, int n)
{
   char * ret_val;
   int i = 0;
   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      while (st[i] != '\n' && st[i] != '\0')
         i++;
      if (st[i] == '\n')
         st[i] = '\0';
      else  // требуется наличие words[i] == '\0'
         while (getchar() != '\n')
            continue;
   }
   return ret_val;
}
```

Вот результаты выполнения этой учебной программы:

```
Какой у вас любимый цветок?
Роза
Роза пахнет как старые валенки.
Какое у вас любимое насекомое?
Комар
Комар пахнет
```

Вы уже могли заметить, что функция `strcat()`, как и `gets()`, может приводить к переполнению буфера. Почему же тогда в стандарте C11 не отказались от `strcat()`, а лишь предложили функцию `strncat()`? Одной из причин может быть то, что функция `gets()` подвергает программу опасности со стороны тех, кто ее использует, в то время как `strcat()` подвергает программу опасности вследствие невнимательности программиста. Будущее поведение того или иного пользователя предвидеть невозможно, но можно контролировать то, что происходит внутри программы. Философия доверия программисту, принятая в C, возлагает на вас ответственность за определение ситуаций, в которых функция `strcat()` может применяться безопасным образом.

Функция `strcmp()`

Предположим, что требуется сравнить введенный кем-то ответ со строкой, хранящейся в памяти, как показано в листинге 11.20.

Листинг 11.20. Программа `nogo.c`

```
/* nogo.c -- будет ли это работать? */
#include <stdio.h>
#define ANSWER "Грант"
#define SIZE 40
char * s_gets(char * st, int n);
int main(void)
{
   char try[SIZE];
   puts("Кто похоронен в могиле Гранта?");
   s_gets(try, SIZE);
   while (try != ANSWER)
   {
      puts("Неправильно! Попытайтесь еще раз.");
      s_gets(try, SIZE);
   }
   puts("Теперь правильно!");
   return 0;
}
char * s_gets(char * st, int n)
{
   char * ret_val;
   int i = 0;
   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      while (st[i] != '\n' && st[i] != '\0')
         i++;
      if (st[i] == '\n')
         st[i] = '\0';
      else  // требуется наличие words[i] == '\0'
         while (getchar() != '\n')
            continue;
   }
   return ret_val;
}
```

Как бы симпатично не выглядела эта программа, работает она некорректно. В действительности ANSWER и try являются указателями, так что сравнение try != ANSWER не проверяет, совпадают ли две строки. Вместо этого оно проверяет, имеют ли указанные две строки один и тот же адрес. Поскольку ANSWER и try хранятся в разных ячейках памяти, их адреса никогда не будут совпадать, и пользователь каждый раз получает сообщение о том, что введенный им ответ неправилен. Такие программы часто сбивают пользователей с толку.

Нам необходима функция, которая сравнивает *содержимое* строк, а не их *адреса*. Можно было бы создать такую функцию самостоятельно, но функция подобного рода уже имеется — strcmp() (от *string comparison* — сравнение строк). Эта функция делает для строк то же, что операции отношений делают для чисел. В частности, она возвращает 0, если оба строковых аргумента одинаковы, и ненулевое значение в противном случае. В листинге 11.21 приведена переделанная программа.

Листинг 11.21. Программа compare.c

```
/* compare.c -- эта программа будет работать */
#include <stdio.h>
#include <string.h>  // объявление strcmp()

#define ANSWER "Грант"
#define SIZE 40
char * s_gets(char * st, int n);

int main(void)
{
   char try[SIZE];

   puts("Кто похоронен в могиле Гранта?");
   s_gets(try, SIZE);
   while (strcmp(try,ANSWER) != 0)
   {
      puts("Неправильно! Попытайтесь еще раз.");
      s_gets(try, SIZE);
   }
   puts("Теперь правильно!");

   return 0;
}

char * s_gets(char * st, int n)
{
   char * ret_val;
   int i = 0;

   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      while (st[i] != '\n' && st[i] != '\0')
         i++;
      if (st[i] == '\n')
         st[i] = '\0';
      else // требуется наличие words[i] == '\0'
         while (getchar() != '\n')
            continue;
   }
   return ret_val;
}
```

На заметку!

Поскольку любое ненулевое значение трактуется как истинное, опытные программисты на C наверняка сократят этот оператор `while` до `while (strcmp(try,ANSWER))`.

Одно из примечательных свойств функции `strcmp()` заключается в том, что она сравнивает строки, а не массивы. Хотя массив `try` занимает 40 ячеек памяти, а строка `"Грант"` – только шесть (одна ячейка отводится под нулевой символ), при сравнении просматривается только часть массива `try` до первого встреченного нулевого символа. Следовательно, `strcmp()` может использоваться для сравнения строк, хранящихся в массивах разных размеров.

А что, если пользователь введет в качестве ответа `"ГРАНТ"`, `"грант"` или `"Улисс С. Грант"`? Программа сообщит о том, что ответ неправильный. Чтобы сделать программу более дружественной, необходимо предусмотреть все возможные правильные ответы. Для этого есть несколько приемов. Например, с помощью директивы `#define` можно определить ответ как `"ГРАНТ"` и написать функцию, которая преобразует все введенные буквы в прописные. Это устраняет проблему употребления прописных букв, но по-прежнему остаются другие формы ответа, о которых следует позаботиться, а также учесть тот факт, что жена Гранта, Джулия, также похоронена в той же могиле. Мы оставляем эти задачи в качестве упражнений для самостоятельной проработки.

Возвращаемое значение функции `strcmp()`

Какое значение возвращает `strcmp()`, если строки не совпадают? В листинге 11.22 показан пример.

Листинг 11.22. Программа `compback.c`

```
/* compback.c -- значения, возвращаемые функцией strcmp() */
#include <stdio.h>
#include <string.h>
int main(void)
{
   printf("strcmp(\"A\", \"A\") возвращает ");
   printf("%d\n", strcmp("A", "A"));

   printf("strcmp(\"A\", \"B\") возвращает ");
   printf("%d\n", strcmp("A", "B"));

   printf("strcmp(\"B\", \"A\") возвращает ");
   printf("%d\n", strcmp("B", "A"));

   printf("strcmp(\"C\", \"A\") возвращает ");
   printf("%d\n", strcmp("C", "A"));

   printf("strcmp(\"Z\", \"a\") возвращает ");
   printf("%d\n", strcmp("Z", "a"));

   printf("strcmp(\"apples\", \"apple\") возвращает ");
   printf("%d\n", strcmp("apples", "apple"));
   return 0;
}
```

В одной из систем вывод может иметь следующий вид:

```
strcmp("A", "A") возвращает 0
strcmp("A", "B") возвращает -1
strcmp("B", "A") возвращает 1
strcmp("C", "A") возвращает 1
strcmp("Z", "a") возвращает -1
strcmp("apples", "apple") возвращает 1
```

Сравнение символа "A" с самим собой возвращает 0. Сравнение "A" и "B" возвращает -1, а обратное сравнение возвращает 1. Из этих результатов следует, что функция strcmp() возвращает отрицательное число, если первая строка в алфавитном порядке предшествует второй, и положительное число, если порядок следования строк противоположный. Таким образом, сравнение "C" с "A" дает 1. Другие системы могут возвратить 2, т.е. разность между значениями ASCII-кодов. Стандарт ANSI требует, чтобы функция strcmp() возвращала отрицательное число, когда первая строка в алфавитном порядке предшествует второй, 0, если строки совпадают, и положительное число, если первая строка в алфавитном порядке следует за второй. Однако точные числовые значения зависят от реализации. Например, ниже приведен вывод для другой реализации, в которой возвращается разность между значениями кодов символов:

```
strcmp("A", "A") возвращает 0
strcmp("A", "B") возвращает -1
strcmp("B", "A") возвращает 1
strcmp("C", "A") возвращает 2
strcmp("Z", "a") возвращает -7
strcmp("apples", "apple") возвращает 115
```

А что, если начальные символы строк идентичны? В общем случае функция strcmp() продвигается по строкам до тех пор, пока не найдет первую пару несовпадающих символов. После этого она возвращает соответствующий код. В самом последнем примере строки "apples" и "apple" совпадают вплоть до финального символа в первой строке. Этот символ сравнивается с шестым символом строки "apple", которым является нулевой символ, имеющий ASCII-код 0. Так как нулевой символ является первым в последовательности ASCII, символ s находится после него, функция возвращает положительное значение.

Последнее сравнение показывает, что strcmp() сравнивает все символы, а не только буквы, поэтому вместо утверждения, что сравнение производится в алфавитном порядке, можно сказать, что strcmp() следует *последовательности сопоставления* машины. Это означает, что символы сравниваются согласно их числовым представлениям, которыми обычно являются значения ASCII-кодов. В кодировке ASCII коды прописных букв предшествуют кодам строчных букв. Таким образом, strcmp("Z", "a") возвращает отрицательное число.

Чаще всего о точном возвращаемом значении можно не беспокоиться. Вполне достаточно знать, нулевое оно или нет, т.е. совпадают ли строки. Если требуется сортировать строки в алфавитном порядке, то нужно знать, каким является результат сравнения — положительным, отрицательным или нулевым.

На заметку!

Функция strcmp() предназначена для сравнения *строк*, а не *символов*. Следовательно, вы можете указывать такие аргументы, как "apples" и "A", но не символьные аргументы наподобие 'A'. Тем не менее, вспомните, что тип char является целочисленным, поэтому для сравнения символов можно применять операции отношений. Предположим, что word — это строка, хранящаяся в массиве элементов char, а ch — переменная типа char. Тогда показанные ниже операторы допустимы:

```
if (strcmp(word, "выход") == 0)    // используйте strcmp() для строк
    puts("Всего хорошего!");
if (ch == 'в')                     // используйте == для символов
    puts("Всего хорошего!");
```

Однако не применяйте ch или 'в' в качестве аргументов для strcmp().

В листинге 11.23 функция `strcmp()` используется для выяснения, когда программа должна остановить чтение ввода.

Листинг 11.23. Программа quit_chk.c

```
/* quit_chk.c -- начало некоторой программы */
#include <stdio.h>
#include <string.h>
#define SIZE 80
#define LIM 10
#define STOP "quit"
char * s_gets(char * st, int n);
int main(void)
{
   char input[LIM][SIZE];
   int ct = 0;
   printf("Введите до %d строк (или quit для завершения):\n", LIM);
   while (ct < LIM && s_gets(input[ct], SIZE) != NULL &&
          strcmp(input[ct],STOP) != 0)
   {
      ct++;
   }
   printf("Введено %d строк(и)\n", ct);
   return 0;
}
char * s_gets(char * st, int n)
{
   char * ret_val;
   int i = 0;
   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      while (st[i] != '\n' && st[i] != '\0')
         i++;
      if (st[i] == '\n')
         st[i] = '\0';
      else  // требуется наличие words[i] == '\0'
         while (getchar() != '\n')
            continue;
   }
   return ret_val;
}
```

Программа завершает чтение входных данных, когда встречает символ EOF (в этом случае `s_gets()` возвращает `NULL`), при вводе слова `quit` или по достижении предела, указанного в `LIM`.

Кстати, чтение входных данных иногда удобнее прекращать путем ввода пустой строки, т.е. нажатием клавиши <Enter> или <Return>, не набирая ничего другого. Для этого оператор цикла `while` можно модифицировать так:

```
while (ct < LIM && s_gets(input[ct], SIZE) != NULL
                && input[ct][0] != '\0')
```

Здесь `input[ct]` представляет собой только что введенную строку, а `input[ct][0]` — первый символ этой строки. Если пользователь вводит пустую строку, то функция `s_gets()` помещает в первый элемент нулевой символ, так что выражение

```
input[ct][0] != '\0'
```

выполняет проверку на предмет пустой входной строки.

Вариация strncmp()

Функция strcmp() сравнивает строки до тех пор, пока не найдет пару соответствующих символов, которые отличаются друг от друга, и этот поиск может продолжаться вплоть до достижения конца одной из строк. Функция strncmp() сравнивает строки до тех пор, пока не обнаружит в них различия либо пока не сравнит количество символов в обеих строках, указанное в третьем аргументе. Например, если необходимо найти строки, начинающиеся с "астро", то поиск можно было бы ограничить первыми пятью символами. В листинге 11.24 показано, как это сделать.

Листинг 11.24. Программа starsrch.c

```
/* starsrch.c -- использование функции strncmp() */
#include <stdio.h>
#include <string.h>
#define LISTSIZE 6
int main()
{
    const char * list[LISTSIZE] =
    {
        "астрономия", "астатизм",
        "астрофизика", "остракизм",
        "астеризм", "астролябия"
    };
    int count = 0;
    int i;

    for (i = 0; i < LISTSIZE; i++)
        if (strncmp(list[i],"astro", 5) == 0)
        {
            printf("Найдено: %s\n", list[i]);
            count++;
        }
    printf("Количество слов в списке,"
           " начинающихся с астро: %d\n", count);

    return 0;
}
```

Вот вывод, полученный из программы:

```
Найдено: астрономия
Найдено: астрофизика
Найдено: астролябия
Количество слов в списке, начинающихся с астро: 3
```

Функции strcpy() и strncpy()

Ранее мы говорили, что если pts1 и pts2 — указатели на строки, то оператор

```
pts2 = pts1;
```

копирует только адрес строки, но не саму строку. Тем не менее, предположим, что вы хотите скопировать строку. В таком случае можете воспользоваться функцией strcpy(). Код в листинге 11.25 предлагает пользователю ввести слова, начинающиеся с буквы *к*. Эта программа копирует ввод во временный массив, и, если первой буквой является *к*, программа использует функцию strcpy() для копирования этой строки из временного файла в место ее постоянного хранения. Функция strcpy() представляет собой строковый эквивалент оператора присваивания.

Листинг 11.25. Программа copy1.c

```
/* copy1.c -- демонстрация использования strcpy() */
#include <stdio.h>
#include <string.h>  // объявление strcpy()
#define SIZE 40
#define LIM 5
char * s_gets(char * st, int n);

int main(void)
{
   char qwords[LIM][SIZE];
   char temp[SIZE];
   int i = 0;

   printf("Введите %d слов, которые начинаются с буквы к:\n", LIM);
   while (i < LIM && s_gets(temp, SIZE))
   {
      if (temp[0] != 'к')
         printf("%s не начинается с буквы к!\n", temp);
      else
      {
         strcpy(qwords[i], temp);
         i++;
      }
   }
   puts("Список принятых слов:");
   for (i = 0; i < LIM; i++)
      puts(qwords[i]);

   return 0;
}

char * s_gets(char * st, int n)
{
   char * ret_val;
   int i = 0;

   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      while (st[i] != '\n' && st[i] != '\0')
         i++;
      if (st[i] == '\n')
         st[i] = '\0';
      else  // требуется наличие words[i] == '\0'
         while (getchar() != '\n')
            continue;
   }
   return ret_val;
}
```

Ниже показаны результаты пробного запуска:

```
Введите 5 слов, которые начинаются с буквы к:
квазар
квота
кристалл
квалификация
больше нет
больше нет не начинается с буквы к!
```

```
конкурс
Список принятых слов:
квазар
квота
кристалл
квалификация
конкурс
```

Обратите внимание, что счетчик i инкрементируется, только когда вводимое слово проходит проверку на наличие первой буквы *к*. Кроме того, в программе применяется проверка на основе символов:

```
if (temp[0] != 'к')
```

Условие выясняет, отличается ли первый символ в массиве temp от буквы *к*. Другая возможность связана с использованием проверки на основе строк:

```
if (strncmp(temp, "к", 1) != 0)
```

Условие определяет, отличаются ли строки temp и "к" друг от друга в первом элементе.

Строка, на которую указывает второй аргумент (temp), копируется в массив, указанный первым аргументом (qword[i]). Копия называется *целью*, а исходная строка – *источником*. Порядок аргументов легко запомнить, т.к. он совпадает с порядком в операторе присваивания (целевая строка находится слева):

```
char target[20];
int x;
x = 50;                              /* присваивание для чисел */
strcpy(target, "Это так!");          /* присваивание для строк */
target = "Очень долго";              /* синтаксическая ошибка  */
```

Ответственность за обеспечение в целевом массиве достаточного пространства для размещения копии источника возлагается на вас. Следующий код приводит к проблеме:

```
char * str;
strcpy(str, "Невозмутимость C");   // проблема
```

Этот вызов strcpy() копирует строку "Невозмутимость C" по адресу, указанному в str, но переменная str не инициализирована, так что копия может оказаться где угодно!

Короче говоря, функция strcpy() принимает в качестве аргументов два указателя на строки. Второй указатель, который ссылается на исходную строку, может быть объявленным указателем, именем массива или строковой константой. Первый указатель, ссылающийся на копию, должен указывать на объект данных, такой как массив с размером, достаточным для хранения этой строки. Вспомните, что объявление массива приводит к выделению пространства под данные, а при объявлении указателя пространство выделяется только для размещения одного адреса.

Другие свойства функции strcpy()

Функция strcpy() обладает еще двумя свойствами, которые вы можете счесть удобными. Во-первых, ее типом является char *. Она возвращает значение своего первого аргумента – адрес символа. Во-вторых, первый аргумент не обязательно должен указывать на начало массива; это позволяет копировать только часть массива. Оба свойства продемонстрированы в листинге 11.26.

Листинг 11.26. Программа copy2.c

```
/* copy2.c -- демонстрация использования strcpy() */
#include <stdio.h>
#include <string.h>  // объявление strcpy()
#define WORDS "наихудшим"
#define SIZE 40

int main(void)
{
    const char * orig = WORDS;
    char copy[SIZE] = "Будьте лучшим, чем могли бы быть.";
    char * ps;

    puts(orig);
    puts(copy);
    ps = strcpy(copy + 7, orig);
    puts(copy);
    puts(ps);

    return 0;
}
```

Ниже показан вывод:

```
наихудшим
Будьте лучшим, чем могли бы быть.
Будьте наихудшим
наихудшим
```

Обратите внимание, что функция `strcpy()` копирует нулевой символ из исходной строки. В этом примере нулевой символ перезаписывает букву е в слове чем внутри copy, так что новая строка заканчивается словом наихудшим (рис. 11.5). Кроме того, ps указывает на восьмой элемент (с индексом 7) массива copy, поскольку в первом аргументе передается copy + 7. Таким образом, вызов `puts(ps)` выводит строку, начиная с этой позиции.

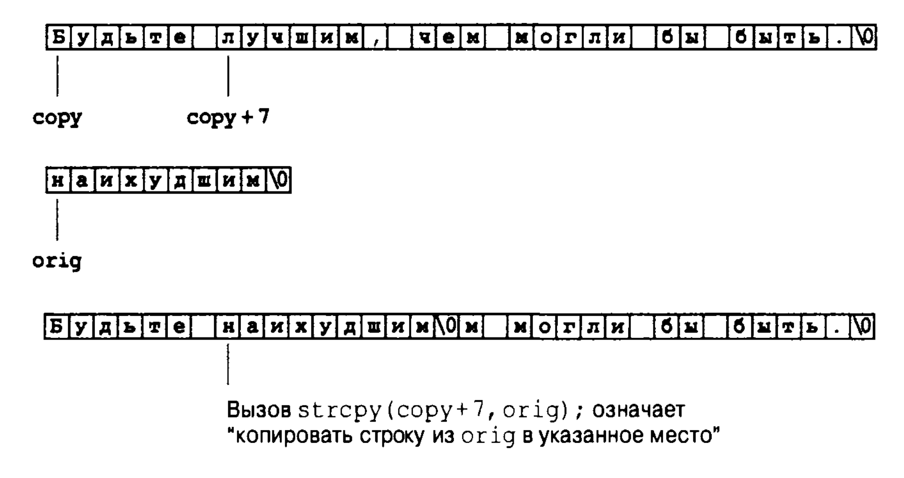


Рис. 11.5. Функция strcpy() использует указатели

Аккуратный выбор: функция strncpy()

Функции strcpy() присуща та же проблема, что и strcat() — ни одна из них не проверяет, умещается ли на самом деле исходная строка в целевую строку. Более безопасный способ копирования строк предусматривает применение функции strncpy(). Эта функция принимает третий аргумент, в котором указывается максимальное количество копируемых символов. В листинге 11.27 приведена переписанная версия кода из листинга 11.25, в которой вместо strcpy() используется strncpy(). Чтобы показать, что происходит в случае, когда размер исходной строки слишком велик, для целевых строк в коде выбран небольшой размер (семь элементов, шесть символов).

Листинг 11.27. Программа copy3.c

```
/* copy3.c -- демонстрация использования strncpy() */
#include <stdio.h>
#include <string.h>  /* объявление strncpy() */
#define SIZE 40
#define TARGSIZE 7
#define LIM 5
char * s_gets(char * st, int n);
int main(void)
{
    char qwords[LIM][TARGSIZE];
    char temp[SIZE];
    int i = 0;
    printf("Введите %d слов, которые начинаются с буквы к:\n", LIM);
    while (i < LIM && s_gets(temp, SIZE))
    {
        if (temp[0] != 'q')
            printf("%s не начинается с буквы к!\n", temp);
        else
        {
            strncpy(qwords[i], temp, TARGSIZE - 1);
            qwords[i][TARGSIZE - 1] = '\0';
            i++;
        }
    }
    puts("Список принятых слов:");
    for (i = 0; i < LIM; i++)
        puts(qwords[i]);
    return 0;
}
char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;
    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else  // требуется наличие words[i] == '\0'
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

Ниже показаны результаты пробного запуска:

```
Введите 5 слов, начинающиеся с буквы q:
квазар
квота
кристалл
квалификация
конкурс
Список принятых слов:
квазар
квота
кристал
квалифи
конкурс
```

Вызов функции `strncpy(target, source, n)` копирует вплоть до `n` символов либо до появления нулевого символа (в зависимости от того, что произойдет раньше) из `source` в `target`. Следовательно, если количество символов в `source` меньше `n`, копируется вся строка, включая нулевой символ. Эта функция никогда не копирует более `n` символов, так что если данный лимит исчерпан до достижения конца исходной строки, то нулевой символ не добавляется. Таким образом, финальный результат может содержать, а может и не содержать нулевой символ. По этой причине значение `n` в программе выбрано на единицу меньше размера целевого массива, а последний элемент массива установлен в нулевой символ:

```
strncpy(qwords[i], temp, TARGSIZE - 1);
qwords[i][TARGSIZE - 1] = '\0';
```

Это обеспечивает сохранение строки. Если исходная строка в действительности умещается в целевую строку, то скопированный вместе с ней нулевой символ помечает настоящий конец строки. Если исходная строка в целевую не умещается, то конец строки помечается последним нулевым символом.

Функция `sprintf()`

Функция `sprintf()` объявлена в заголовочном файле `stdio.h`, а не в `string.h`. Она работает подобно `printf()`, но осуществляет запись в строку, а не на дисплей. Таким образом, она предоставляет способ объединения нескольких элементов в единую строку. Первый аргумент `sprintf()` — это адрес целевой строки. Остальные аргументы аналогичны аргументам в `printf()` — строка спецификации преобразования и список элементов, предназначенных для записи.

В листинге 11.28 функция `sprintf()` применяется для объединения трех элементов (двух строк и числа) в одну строку. Обратите внимание, что `sprintf()` используется так же, как это бы делалось в случае функции `printf()`, кроме того, что результирующая строка сохраняется в массиве `formal`, а не отображается на экране.

Листинг 11.28. Программа `format.c`

```
/* format.c -- форматирование строки */
#include <stdio.h>
#define MAX 20
char * s_gets(char * st, int n);

int main(void)
{
    char first[MAX];
    char last[MAX];
```

```
    char formal[2 * MAX + 10];
    double prize;

    puts("Введите свое имя:");
    s_gets(first, MAX);
    puts("Введите свою фамилию:");
    s_gets(last, MAX);
    puts("Введите сумму денежного приза:");
    scanf("%lf", &prize);
    sprintf(formal, "%s, %-19s: $%6.2f\n", last, first, prize);
    puts(formal);

    return 0;
}
char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else  // требуется наличие words[i] == '\0'
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

Ниже показаны результаты пробного запуска:

```
Введите свое имя:
Остап
Введите свою фамилию:
Бендер
Введите сумму денежного приза:
25000
Бендер, Остап             : $25000.00
```

Функция `sprintf()` принимает входные данные и форматирует его в стандартном виде, после чего сохраняет в строку `formal`.

Другие строковые функции

Библиотека ANSI C содержит более 20 функций, предназначенных для работы со строками, и далее приведены краткие описания наиболее часто применяемых из них.

- `char *strcpy(char * restrict s1, const char * restrict s2);`

  Эта функция копирует строку (включая нулевой символ), указанную `s2`, в ячейку, на которую указывает `s1`. Возвращаемым значением является `s1`.

- `char *strncpy(char * restrict s1, const char * restrict s2, size_t n);`

  Эта функция копирует в ячейку, указанную `s1`, не более `n` символов из строки, на которую указывает `s2`. Возвращаемым значением является `s1`. Символы, следующие за нулевым символом, не копируются, и если исходная строка короче `n`

символов, оставшаяся часть целевой строки заполняется нулевыми символами. Если исходная строка содержит n или больше символов, нулевой символ не копируется. Возвращаемым значением является s1.

- `char *strcat(char * restrict s1, const char * restrict s2);`

  Строка, указанная s2, копируется в конец строки, на которую указывает s1. Первый символ строки s2 копируется поверх нулевого символа строки s1. Возвращаемым значением является s1.

- `char *strncat(char * restrict s1, const char * restrict s2, size_t n);`

  К строке s1 добавляется не более n символов строки s2, причем первый символ строки s2 копируется поверх нулевого символа строки s1. Нулевой символ и любые другие символы, которые за ним следуют в строке s2, не копируются, а к результату добавляется нулевой символ. Возвращаемым значением является s1.

- `int strcmp(const char * s1, const char * s2);`

  Эта функция возвращает положительное значение, если в последовательности сопоставления машины строка s1 следует за строкой s2, значение 0, если строки идентичны, и отрицательное значение, если в последовательности сопоставления первая строка предшествует второй.

- `int strncmp(const char * s1, const char * s2, size_t n);`

  Эта функция работает подобно `strcmp()`, за исключением того, что процедура сравнения останавливается после просмотра n символов либо при появлении первого нулевого символа, в зависимости от того, что произойдет раньше.

- `char *strchr(const char * s, int c);`

  Эта функция возвращает указатель на первую ячейку строки s, в которой содержится символ c. (Завершающий нулевой символ является частью строки, так что его тоже можно искать.) Если символ не найден, функция возвращает нулевой указатель.

- `char *strpbrk(const char * s1, const char * s2);`

  Эта функция возвращает указатель на первую ячейку строки s1, в которой содержится любой символ, найденный в строке s2. Эта функция возвращает нулевой указатель, если ни одного символа не найдено.

- `char *strrchr(const char * s, int c);`

  Эта функция возвращает указатель на последнее вхождение символа c в строке s. (Завершающий нулевой символ является частью строки, так что его тоже можно искать.) Если символ не найден, функция возвращает нулевой указатель.

- `char *strstr(const char * s1, const char * s2);`

  Эта функция возвращает указатель на первое вхождение строки s2 внутри строки s1. Если строка не найдена, функция возвращает нулевой указатель.

- `size_t strlen(const char * s);`

  Эта функция возвращает количество символов, не включая нулевой, находящихся в строке s.

Обратите внимание, что во всех прототипах используется ключевое слово `const`, чтобы отразить, какие строки не изменяются функцией. Например, взгляните на следующий прототип:

```
char *strcpy(char * restrict s1, const char * restrict s2);
```

Это означает, что s2 указывает на строку, которая не может быть изменена, во всяком случае, функцией strcpy(), но s1 указывает на строку, изменять которую разрешено. В этом есть смысл, т.к. s1 — целевая строка, подвергающаяся изменениям, а s2 — исходная строка, которая должна оставаться неизменной.

Ключевое слово restrict, обсуждаемое в главе 12, устанавливает ограничения на то, как должны применяться аргументы функции, например, оно указывает на недопустимость копирования строки в саму себя.

Тип size_t, как отмечалось в главе 5 — это любой тип, возвращаемый операцией sizeof. В языке С заявлено, что операция sizeof возвращает целочисленный тип, но не задано, какой именно; таким образом, в одной системе size_t может быть unsigned int, а в другой — unsigned long. В заголовочном файле string.h тип имеется определение size_t для конкретной системы либо указывается ссылка на другой заголовочный файл, содержащий необходимое определение.

Как уже упоминалось, в разделе V приложения Б перечислены все функции в семействе string.h. Многие реализации предоставляют дополнительные функции помимо требуемых стандартом ANSI. Чтобы посмотреть, что доступно, обращайтесь в документацию по своей реализации.

Рассмотрим простой случай использования одной из таких функций. Ранее было показано, что функция fgets() при чтении строки ввода сохраняет символ новой строки в целевой строке. В нашей функции s_gets() для обнаружения символа новой строки применялся цикл while, но вместо него можно использовать strchr(). Сначала найдите с помощью функции strchr() символ новой строки, если он есть. Когда он обнаруживается, strchr() возвращает адрес символа новой строки, и затем по этому адресу можно поместить нулевой символ:

```
fgets(line, 80, stdin);
find = strchr(line, '\n');      // поиск символа новой строки
if (find)                       // если адрес не является NULL,
    *find = '\0';               // поместить туда нулевой символ
```

Если strchr() не удается найти символ новой строки, функция fgets() достигнет лимита на размер еще до конца строки. Для обработки такой ситуации к оператору if можно добавить конструкцию else, как это делалось в s_gets().

Давайте рассмотрим завершенную программу, которая обрабатывает строки.

Пример обработки строк: сортировка строк

Давайте решим практическую задачу сортировки строк в алфавитном порядке. Эта задача возникает при подготовке списков фамилий, во время индексации и во многих других ситуациях. Одним из основных инструментов в такой программе является функция strcmp(), поскольку она может применяться для выяснения порядка следования двух строк. Генеральный план предполагает чтение массива строк, их сортировку и вывод. Ранее мы представляли схему для чтения строк, и с этого мы начнем настоящую программу. С выводом строк никаких проблем не связано. Мы будет использовать один из стандартных алгоритмов сортировки, который объясним позже. Кроме того, мы применим также несколько необычный прием; посмотрите, сможете ли вы самостоятельно обнаружить его. Программа показана в листинге 11.29.

Листинг 11.29. Программа sort_str.c

```
/* sort_str.c -- считывает строки и сортирует их */
#include <stdio.h>
#include <string.h>
```

```
#define SIZE 81                         /* лимит на длину строки, включая \0     */
#define LIM 20                          /* максимальное количество читаемых строк */
#define HALT ""                         /* нулевая строка для прекращения ввода   */
void stsrt(char *strings[], int num);   /* функция сортировки строк */
char * s_gets(char * st, int n);

int main(void)
{
   char input[LIM][SIZE];               /* массив для сохранения входных данных */
   char *ptstr[LIM];                    /* массив переменных типа указателя    */
   int ct = 0;                          /* счетчик ввода                      */
   int k;                               /* счетчик вывода                     */

   printf("Введите до %d строк, и они будут отсортированы.\n",LIM);
   printf("Чтобы остановить ввод, нажмите клавишу Enter в начале строки.\n");
   while (ct < LIM && s_gets(input[ct], SIZE) != NULL
         && input[ct][0] != '\0')
   {
      ptstr[ct] = input[ct];  /* установка указателей на строки */
      ct++;
   }
   stsrt(ptstr, ct);          /* сортировщик строк            */
   puts("\nОтсортированный список:\n");
   for (k = 0; k < ct; k++)
      puts(ptstr[k]);         /* отсортированные указатели     */

   return 0;
}
/* функция сортировки указателей на строки */
void stsrt(char *strings[], int num)
{
   char *temp;
   int top, seek;
   for (top = 0; top < num-1; top++)
      for (seek = top + 1; seek < num; seek++)
         if (strcmp(strings[top],strings[seek]) > 0)
         {
            temp = strings[top];
            strings[top] = strings[seek];
            strings[seek] = temp;
         }
}
char * s_gets(char * st, int n)
{
   char * ret_val;
   int i = 0;
   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      while (st[i] != '\n' && st[i] != '\0')
         i++;
      if (st[i] == '\n')
         st[i] = '\0';
      else  // требуется наличие words[i] == '\0'
         while (getchar() != '\n')
            continue;
   }
   return ret_val;
}
```

Для тестирования программы из листинга 11.29 мы ввели отрывок из поэмы А. С. Пушкина "Руслан и Людмила":

```
Введите до 20 строк, и они будут отсортированы.
Чтобы остановить ввод, нажмите клавишу Enter в начале строки.
У лукоморья дуб зеленый;
Златая цепь на дубе том:
И днем и ночью кот ученый
Все ходит по цепи кругом;
Отсортированный список:
Все ходит по цепи кругом;
Златая цепь на дубе том:
И днем и ночью кот ученый
У лукоморья дуб зеленый;
```

Похоже, упорядочение строк по алфавиту сказалось на стихотворении довольно забавным образом.

Сортировка указателей вместо строк

Сложная часть этой программы связана с тем, что вместо самих строк переупорядочиваются *указатели* на эти строки. Давайте посмотрим, что это означает. Первоначально элемент ptrst[0] установлен в input[0] и т.д. В результате указатель ptrst[i] ссылается на первый символ массива input[i]. Каждый элемент input[i] представляет собой массив из 81 элемента, а каждый элемент ptrst[i] – отдельную переменную. Процедура сортировки переупорядочивает ptrst, оставляя input незатронутым. Если, например, в алфавитном порядке input[1] находится перед input[0], то программа переключит указатели в ptrst, из-за чего ptrst[0] будет ссылаться на начало input[1], а ptrst[1] – на начало input[0]. Это намного проще, чем использование функции strcpy() для обмена содержимым двух строк в input. На рис. 11.6 представлена еще одна точка зрения на этот процесс. Данный процесс также обладает тем преимуществом, что сохраняет первоначальный порядок в массиве input.

Перед сортировкой:
ptrst[0] указывает на input[0], ptrst[1] указывает на input[1] и т. д.

input[0] ▶ | У | | л | у | ... |
[0][0] [0][1] ... [0][80]

input[1] ▶ | З | л | а | т | ... |
[1][0] [1][1] ... [1][80]

input[2] ▶ | И | | д | н | ... |
[2][0] [2][1] ... [2][80]

input[3] ▶ | В | с | е | | ... |
[3][0] [3][1] ... [3][80]

После сортировки:
ptrst[0] указывает на input[3], ptrst[1] указывает на input[1] и т. д.

Рис. 11.6. Сортировка указателей на строки

Алгоритм сортировки выбором

Для сортировки указателей мы применяем алгоритм *сортировки выбором*. Идея заключается в использовании цикла for для сравнения всех элементов по очереди с первым. Если сравниваемый элемент предшествует текущему первому элементу, они меняются местами. К моменту, когда достигается конец цикла, первый элемент содержит указатель на строку, находящуюся первой в последовательности сопоставления машины. Затем внешний цикл for повторяет процесс, начиная на этот раз со второго элемента input. Когда внутренний цикл завершится, во втором элементе ptrst окажется указатель на строку, находящуюся второй в последовательности сопоставления. Процесс продолжается до тех пор, пока не будут отсортированы все элементы.

Давайте более подробно рассмотрим сортировку выбором. Ниже показан ее набросок в псевдокоде:

```
для n = первый до n = предпоследний элемент
    найти наибольшее из оставшихся чисел и поместить его в n-й элемент
```

Это работает следующим образом. Начините с n = 0. Просмотрите весь массив, найдите наибольшее число и поменяйте его и первый элемент местами. Далее установите n = 1 и просмотрите все элементы массива кроме первого. Найдите наибольший из оставшихся элементов и поменяйте его и второй элемент местами. Продолжайте этот процесс до тех пор, пока не достигнете предпоследнего элемента. Теперь остались только два элемента. Сравните их и поместите больший в позицию предпоследнего элемента. В итоге наименьший элемент занял свою окончательную позицию.

Выглядит так, что это задача для цикла for, но мы еще должны более подробно описать процесс "найти и поместить". Один из способов выбора наибольшего значения из числа оставшихся предполагает сравнение первого и второго элементов в оставшейся части массива. Если второй элемент больше первого, выполните обмен их значениями. Далее сравните первый элемент с третьим. Если третий элемент больше, произведите обмен их значениями. Каждый обмен приводит к перемещению большего элемента ближе к началу списка. Продолжайте действовать в подобной манере до тех пор, пока не произойдет сравнение первого элемента с последним. После завершения наибольшее значение окажется в первом элементе оставшегося массива. Итак, вы отсортировали массив для первого элемента, однако остальные элементы находятся в беспорядке. Вот как можно представить процедуру с помощью псевдокода:

```
для n = предпредпоследний элемент
    сравнить n-й элемент с первым элементом;
    если n-й элемент больше, выполнить обмен их значениями
```

Этот процесс выглядит как еще один цикл for. Он должен быть вложен в первый цикл for. Внешний цикл указывает, какой элемент массива должен быть заполнен, а внутренний цикл находит значение, которое в него следует поместить. Объединив вместе обе части псевдокода и переведя его на С, мы получаем функцию, показанную в листинге 11.29. Кстати, в библиотеке С имеется более совершенная функция сортировки по имени qsort(). Помимо прочего она принимает указатель на функцию, выполняющую сравнение при сортировке. Ее работа будет продемонстрирована в главе 16.

Символьные функции ctype.h и строки

В главе 7 было представлено семейство функций обработки символов ctype.h. Эти функции не могут быть применены к строке как единому целому, но могут использоваться с отдельными символами в строке. В листинге 11.30 определена функ-

ция, которая применяет toupper() к каждому символу строки, преобразуя символы всей строки в верхний регистр. В нем также определена функция, которая использует ispunct() для подсчета знаков препинания в строке. Наконец, здесь применяется функция strchr(), как было описано ранее, для обработки символа новой строки, если таковой присутствует, в строке, прочитанной с помощью fgets().

Листинг 11.30. Программа mod_str.c

```
/* mod_str.c -- модифицирует строку */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define LIMIT 81
void ToUpper(char *);
int PunctCount(const char *);

int main(void)
{
   char line[LIMIT];
   char * find;

   puts("Введите строку:");
   fgets(line, LIMIT, stdin);
   find = strchr(line, '\n');       // поиск символа новой строки
   if (find)                        // если адрес не является NULL,
      *find = '\0';                 // поместить туда нулевой символ
   ToUpper(line);
   puts(line);
   printf("Эта строка содержит %d знаков препинания.\n",
        PunctCount(line));

   return 0;
}

void ToUpper(char * str)
{
   while (*str)
   {
      *str = toupper(*str);
      str++;
   }
}

int PunctCount(const char * str)
{
   int ct = 0;
   while (*str)
   {
      if (ispunct(*str))
         ct++;
      str++;
   }

   return ct;
}
```

Цикл while (*str) обрабатывает каждый символ в строке, на которую указывает str, пока не будет достигнут нулевой символ. В этот момент *str становится равным 0 (код нулевого символа), или ложному значению, и цикл прекращается.

Вот результаты пробного запуска:

```
Введите строку:
Спокойно, спокойно. За дело берусь я. Заседание продолжается.
СПОКОЙНО, СПОКОЙНО. ЗА ДЕЛО БЕРУСЬ Я. ЗАСЕДАНИЕ ПРОДОЛЖАЕТСЯ.
Эта строка содержит 4 знаков препинания.
```

Функция ToUpper() применяет toupper() к каждому символу строки. (Поскольку в С делается различие между символами верхнего и нижнего регистра, эти имена функций считаются разными.) Как определено в стандарте ANSI С, функция toupper() изменяет только символы, представленные в нижнем регистре. Тем не менее, в очень старых реализациях С эта проверка не выполнялась автоматически, так что в раннем коде обычно поступали следующим образом:

```
if (islower(*str))   /* до выхода стандарта ANSI C --
                        проверка перед преобразованием */
   *str = toupper(*str);
```

Кстати, функции из ctype.h обычно реализованы как *макросы*. Они представляют собой конструкции препроцессора С, которые действуют подобно функциям, но имеют несколько важных отличий. Мы раскроем макросы в главе 16.

В этой программе используется сочетание функций fgets() и strchr() для чтения строки ввода и замены символа новой строки нулевым символом. Главное отличие между этим подходом и применением функции s_gets() заключается в том, что s_gets() отбрасывает оставшиеся символы строки ввода (если они есть), подготавливая программу к обработке следующего оператора ввода. В данном случае существует только один оператор ввода, поэтому такой дополнительный шаг не нужен.

Давайте восполним еще один пробел в наших знаниях и рассмотрим void внутри круглых скобок в функции main().

Аргументы командной строки

До появления современного графического интерфейса существовал интерфейс командной строки. Примерами могут служить DOS и Unix, к тому же терминал Linux предоставляет Unix-подобную среду командной строки. *Командная строка* — это место, где вы вводите с клавиатуры информацию для запуска своей программы в среде командной строки. Предположим, что программа хранится в файле по имени fuss. Тогда командная строка для ее запуска в среде Unix может выглядеть так:

```
$ fuss
```

А вот ее вид для режима командной строки Windows:

```
C> fuss
```

Аргументы командной строки — это дополнительные элементы в той же самой строке, например:

```
$ fuss -r Ginger
```

Программа на С может считывать эти дополнительные элементы для собственных целей (рис. 11.7).

Программа на С читает эти элементы, используя аргументы функции main(). В листинге 11.31 приведен типичный пример.



Рис. 11.7. Аргументы командной строки

Листинг 11.31. Программа repeat.c

```
/* repeat.c -- функция main() с аргументами */
#include <stdio.h>
int main(int argc, char *argv[])
{
   int count;

   printf("Количество аргументов, указанных в командной строке: %d\n", argc - 1);
   for (count = 1; count < argc; count++)
      printf("%d: %s\n", count, argv[count]);
   printf("\n");

   return 0;
}
```

Скомпилируйте программу в исполняемый файл repeat. Вот что происходит, когда вы запускаете ее в командной строке:

```
C>repeat Все будет хорошо.
Количество аргументов, указанных в командной строке: 3
1: Все
2: будет
3: хорошо
```

Вы уже поняли, почему программа называется repeat ("повторить"), и наверняка хотите узнать, как она работает. Ниже приведены необходимые пояснения.

Компиляторы C позволяют функции main() не принимать аргументов либо иметь два аргумента. (Некоторые реализации разрешают принимать дополнительные аргументы, но это считается расширением стандарта.) В случае двух аргументов первым является количество элементов в командной строке. По традиции (хотя и не обязательно) этот аргумент типа int называется argc (от *argument count* – количество аргументов). Для выяснения, где заканчивается один элемент и начинается другой, система применяет пробелы. Таким образом, в запуске repeat присутствуют четыре элемента, включая имя команды, а в запуске fuss – три. Программа сохраняет элементы командной строки в памяти и помещает адреса элементов в массив указателей. Адрес этого массива сохраняется во втором аргументе. По соглашению этот указатель

на указатели имеет имя argv (от *argument values* – значения аргументов). Когда возможно (в некоторых операционных системах это не разрешено), элементу argv[0] присваивается имя самой программы. Затем элементу argv[1] присваивается первый из следующих далее элементов командной строки и т.д. В рассматриваемом примере мы имеем следующие отношения:

argv[0] указывает на repeat (в большинстве систем)

argv[1] указывает на Все

argv[2] указывает на будет

argv[3] указывает на хорошо

В листинге 11.31 с помощью цикла for элементы командной строки выводятся по очереди. Вспомните, что спецификатор %s для printf() ожидает предоставления в аргументе адреса строки. Каждый элемент – argv[0], argv[1] и т.д. – является таким адресом.

Данная форма аналогична форме любой другой функции, принимающей формальные аргументы. Многие программисты используют другое объявление для argv:

```
int main(int argc, char **argv)
```

Такое альтернативное объявление argv в действительности эквивалентно char *argv[]. Оно говорит о том, что argv представляет собой указатель на указатель на char. Рассматриваемый пример сводится к тому же. В нем содержится массив из семи элементов. Имя массива – это указатель на первый элемент, так что argv указывает на argv[0], a argv[0] является указателем на char. Следовательно, даже в исходном определении argv будет указателем на указатель на char. Вы можете применять любую форму, однако мы полагаем, что первая форма более ясно показывает, что argv представляет набор строк.

Многие среды, включая Unix и DOS, позволяют использовать кавычки для объединения нескольких слов в один аргумент. Например, команда

```
repeat "Я был здесь" давно
```

присваивает строку "Я был здесь" элементу argv[1], а строку "давно" – элементу argv[2].

Аргументы командной строки в интегрированных средах

В интегрированных средах Windows, таких как Apple XCode, Microsoft Visual C++ и Embarcadero C++ Builder, для запуска программ командная строка не применяется. Тем не менее, в ряде сред предлагается диалоговое окно свойств проекта, которое позволяет указывать аргументы командной строки для конкретного проекта. В других случаях может быть возможность компиляции программы в IDE-среде, и затем открытия окна MS-DOS для запуска программы в режиме командной строки. Но задача упрощается, если система позволяет запускать компилятор командной строки, такой как GCC.

Аргументы командной строки в Macintosh

Если вы пользуетесь XCode 4.6 (или аналогичной версией), то для предоставления аргументов командной строки выберите в меню Products (Продукты) пункт Scheme⇨Edit Scheme⇨Run (Схема⇨Изменить схему⇨Выполнить). В открывшемся диалоговом окне перейдите на вкладку Arguments (Аргументы) и введите аргументы в поле Arguments Pass on Launch (Аргументы, передаваемые при запуске).

Или же можно переключиться в режим Terminal (Терминал), получив доступ к среде командной строки Unix. Затем можно либо перейти в каталог (так в Unix называют папку), содержащий исполняемый код программы, либо, если вы загрузили инструменты командной строки, воспользоваться компилятором gcc или clang, чтобы скомпилировать программу.

Преобразования строк в числа

Числа могут храниться либо как строки, либо в числовой форме. Хранение числа в виде строки означает хранение символов для цифр. Например, число 213 может быть сохранено в массиве символьной строки как цифры '2', '1', '3', '\0'. Хранение 213 в числовой форме означает его хранение как значения, скажем, типа int.

Числовые формы в C требуются для числовых операций, таких как сложение и сравнение, но отображение чисел на экране требует строковой формы, поскольку экран воспроизводит символы. Функции printf() и sprintf() посредством %d и других спецификаторов преобразуют числовые формы в строковые, а функция scanf() может преобразовывать вводимые строки в числовые формы. В C также имеются функции, единственным назначением которых является преобразование строковых форм в числовые.

Предположим, к примеру, что вам нужна программа для работы с аргументом командной строки. К сожалению, аргументы командной строки читаются как строки. Вследствие этого, чтобы иметь дело с числовым значением, сначала вы должны преобразовать полученную строку в число. Если число целое, можете применить функцию atoi() (от *alphanumeric to integer* – преобразование алфавитно-цифрового значения в целое число). Эта функция принимает строку в качестве аргумента и возвращает соответствующее целочисленное значение. В листинге 11.32 приведен пример использования.

Листинг 11.32. Программа hello.c

```
/* hello.c -- преобразует аргумент командной строки в число */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int i, times;

    if (argc < 2 || (times = atoi(argv[1])) < 1)
        printf("Использование: %s положительное-число\n", argv[0]);
    else
        for (i = 0; i < times; i++)
            puts("Хорошего дня!");

    return 0;
}
```

Вот результаты пробного запуска:

```
$ hello 3
Хорошего дня!
Хорошего дня!
Хорошего дня!
```

Символ $ – это приглашение в Unix и Linux. (В некоторых системах Unix применяется %.) Аргумент командной строки 3 был сохранен как строка 3\0.

Функция atoi() преобразует эту строку в целочисленное значение 3, которое присваивается переменной times. Затем times задает количество итераций цикла for.

Если вы запустите эту программу без аргумента командной строки, проверка argc < 2 приведет к ее прекращению и выводу сообщения о способе использования программы. То же самое происходит, если переменная times равна 0 либо имеет отрицательное значение. Правило вычисления логических операций в С гарантирует, что при argc < 2 оценка atoi(argv[1]) не производится.

Функция atoi() по-прежнему работает, если строка только начинается с целого числа. В таком случае она преобразовывает символы до тех пор, пока не встретит символ, не являющийся частью целого числа. Например, atoi("42regular") возвращает целое число 42. А что, если командной строкой будет hello what? В нашей реализации С функция atoi() возвращает значение 0, если ее аргумент не распознается как числовой. Тем не менее, в стандарте С указано, что поведение в данном случае не определено. Функция strtol(), которая вскоре будет обсуждаться, обеспечивает более надежный контроль ошибок.

В программу включен заголовочный файл stdlib.h, потому что, начиная с ANSI С, он содержит объявление функции atoi(). В этом заголовочном файле также объявлены функции atof() и atol(). Функция atof() преобразует строку в значение типа double, а функция atol() — в значение типа long. Они работают аналогично atoi(), и поэтому имеют, соответственно, тип double и тип long.

Стандарт ANSI С предоставляет усложненные версии этих функций: strtol() преобразует строку в значение типа long, strtoul() — типа unsigned long, а strtod() — типа double. Более сложный аспект упомянутых функций связан с тем, что они идентифицируют и сообщают о первом символе в строке, который не относится к числу. Кроме того, функции strtol() и strtoul() позволяют указывать основание системы счисления.

Давайте рассмотрим пример, в котором задействована функция strtol(). Ее прототип имеет следующий вид:

```
long strtol(const char * restrict nptr, char ** restrict endptr, int base);
```

Здесь nptr — указатель на строку, подлежащую преобразованию, endptr — адрес указателя, который устанавливается в адрес символа, прекращающего ввод числа, и base — основание системы счисления, в которой записано число. Это поможет прояснить пример, приведенный в листинге 11.33.

Листинг 11.33. Программа strcnvt.c

```
/* strcnvt.c -- использование функции strtol() */
#include <stdio.h>
#include <stdlib.h>
#define LIM 30
char * s_gets(char * st, int n);

int main()
{
   char number[LIM];
   char * end;
   long value;

   puts("Введите число (или пустую строку для выхода из программы):");
   while(s_gets(number, LIM) && number[0] != '\0')
   {
      value = strtol(number, &end, 10); /* по основанию 10 */
      printf("десятичный ввод, десятичный вывод: %ld, прекращен на %s (%d)\n",
             value, end, *end);
```

```
      value = strtol(number, &end, 16); /* по основанию 16 */
      printf("шестнадцатеричный ввод, шестнадцатеричный вывод: %ld, прекращен на %s (%d)\n",
            value, end, *end);
      puts("Следующее число:");
   }
   puts("Программа завершена.\n");

   return 0;
}

char * s_gets(char * st, int n)
{
   char * ret_val;
   int i = 0;

   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      while (st[i] != '\n' && st[i] != '\0')
         i++;
      if (st[i] == '\n')
         st[i] = '\0';
      else  // требуется наличие words[i] == '\0'
         while (getchar() != '\n')
            continue;
   }
   return ret_val;
}
```

Ниже показаны результаты пробного запуска:

```
Введите число (или пустую строку для выхода из программы):
10
десятичный ввод, десятичный вывод: 10, прекращен на  (0)
шестнадцатеричный ввод, шестнадцатеричный вывод: 16, прекращен на  (0)
Следующее число:
10atom
десятичный ввод, десятичный вывод: 10, прекращен на atom (97)
шестнадцатеричный ввод, шестнадцатеричный вывод: 266, прекращен на atom (116)
Следующее число:

Программа завершена.
```

Прежде всего, обратите внимание, что строка "10" преобразуется в число 10, когда base равно 10, и в 16, когда base равно 16. Кроме того, если end указывает на символ, то *end — это сам символ. Следовательно, первое преобразование завершается, когда достигнут нулевой символ, поэтому end указывает на нулевой символ. Таким образом, вывод end приводит к отображению пустой строки, а вывод *end с форматом %d — к отображению ASCII-кода нулевого символа.

Для второй строки ввода (в интерпретации с десятичным представлением) указатель end получает адрес символа 'a'. Тогда вывод указателя end приводит к отображению строки "atom", а вывод *end — к отображению ASCII-кода символа 'a'. Однако, как только base изменяется на 16, символ 'a' распознается как допустимая шестнадцатеричная цифра, и функция преобразует шестнадцатеричное число 10a в десятичное 266.

Функция strtol() работает с системами счисления, имеющими основание вплоть до 36, применяя в качестве цифр все буквы английского алфавита до 'z'. Функция strtoul() делает то же самое, но преобразует значения без знака.

Функция `strtod()` работает только в десятичной системе счисления, так что она принимает только два аргумента.

Многие реализации располагают функциями `itoa()` и `ftoa()`, предназначенными для преобразования целочисленных значений и значений с плавающей запятой в строки. Тем не менее, они не являются частью стандартной библиотеки C, поэтому для большей совместимости используйте вместо них `sprintf()`.

Ключевые понятия

Многие программы имеют дело с текстовыми данными. Программа может предложить ввести ваше имя, список корпораций, адрес, ботаническое название папоротника, музыкальное сопровождение и т.п.; поскольку мы взаимодействуем с окружающим миром посредством слов, примерам применения текста на самом деле нет конца. Строки являются средством, используемым в программах на C для обработки текста.

Строка в C, независимо от того, как она идентифицирована – символьным массивом, указателем или строковым литералом, – хранится в виде последовательности байтов, содержащих коды символов, и эта последовательность завершается нулевым символом. Учитывая удобство строк, в C предоставлена библиотека функций для манипулирования, поиска и анализа строк. В частности, имейте в виду, что при сравнении строк вы должны применять функцию `strcmp()`, а не операции отношений, а для присваивания строки символьному массиву – функцию `strcpy()` или `strncpy()` вместо операции присваивания.

Резюме

В языке C *строка* – это последовательность значений типа `char`, завершающаяся нулевым символом, `'\0'`. Строка может также храниться в символьном массиве. Кроме того, строка может быть представлена с помощью *строковой константы*, в которой символы кроме нулевого заключены в двойные кавычки. Нулевой символ предоставляется компилятором. Таким образом, строка `"мир"` сохраняется в памяти как последовательность из четырех символов: м, и, р и `\0`. Длина строки, измеренная посредством `strlen()`, не учитывает нулевой символ.

Строковые константы, также известные как *строковые литералы*, могут использоваться для инициализации символьных массивов. Размер массива должен быть, по меньшей мере, на единицу больше длины строки, чтобы можно было включить нулевой символ. Строковые константы также могут применяться для инициализации указателей на тип `char`.

Для идентификации обрабатываемой строки функции используют указатель на первый символ этой строки. Обычно соответствующим фактическим аргументом является имя массива, переменная типа указателя или строка в двойных кавычках. В каждой ситуации передается адрес первого символа. В общем случае передача длины строки не обязательна, т.к. для обнаружения конца строки функция может применять завершающий нулевой символ.

Функция `fgets()` извлекает строку из ввода, а функции `puts()` и `fputs()` отображают строку вывода. Они входят в семейство функций `stdio.h`, как раньше было с теперь уже устаревшей и постепенно выводимой из употребления функцией `gets()`.

Библиотека C содержит несколько функций *обработки строк*. В стандарте ANSI C эти функции объявлены в файле `string.h`. Библиотека также имеет ряд функций обработки символов, которые объявлены в файле `ctype.h`.

Программе можно предоставить доступ к *аргументам командной строки* с помощью двух формальных аргументов функции main(). Первый аргумент, по традиции называемый argc, имеет тип int, и ему присваивается количество слов в командной строке. Второй аргумент, традиционно имеющий имя argv, представляет собой указатель на массив указателей на тип char. Каждый указатель на char ссылается на один из элементов командной строки, при этом argv[0] указывает на имя команды, argv[1] — на первый аргумент командной строки, argv[2] — на второй аргумент и т.д.

Функции atoi(), atol() и atof() преобразуют строковые представления чисел в значения типов int, long и double. Функции strtol(), strtoul() и strtod() преобразуют строковые представления чисел в формы типов long, unsigned long и double.

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля приведены в приложении А.

1. Какая ошибка допущена в представленном объявлении символьной строки?

```
int main(void)
{
   char name[] = {'Л', 'у', 'н', 'а'};
   ...
}
```

2. Что выведет следующая программа?

```
#include <stdio.h>
int main(void)
{
   char note[] = "Увидимся завтра в кафе.";
   char *ptr;

   ptr = note;
   puts(ptr);
   puts(++ptr);
   note[7] = '\0';
   puts(note);
   puts(++ptr);
   return 0;
}
```

3. Что выведет следующая программа?

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char food[] = "Вкусно";
   char *ptr;

   ptr = food + strlen(food);
   while (--ptr >= food)
      puts(ptr);
   return 0;
}
```

4. Что выведет следующая программа?

```
#include <stdio.h>
#include <string.h>
```

```
int main(void)
{
   char goldwyn[40] = "огу я смог ";
   char samuel[40] = "За всю дор";
   const char * quote = "осилить лишь часть.";

   strcat(goldwyn, quote);
   strcat(samuel, goldwyn);
   puts(samuel);

   return 0;
}
```

5. Приведенные ниже вопросы позволят попрактиковаться со строками, циклами, указателями и их инкрементированием. Предположим, что имеется такое определение функции:

```
#include <stdio.h>
char *pr (char *str)
{
   char *pc;

   pc = str;
   while (*pc)
     putchar(*pc++);
   do {
     putchar(*--pc);
     } while (pc - str);
   return (pc);
}
```

Взгляните на следующий вызов функции:

```
x = pr("Xo Xo Xo!");
```

а. Что он выводит?

б. Какой тип должна иметь переменная x?

в. Какое значение получает переменная x?

г. Что означает выражение `*--pc`, и чем оно отличается от `--*pc` ?

д. Что будет выведено, если `*--pc` заменить выражением `*pc--`?

е. Для чего предназначены два выражения проверки в операторах `while`?

ж. Что произойдет, если функции `pr()` передать в качестве аргумента пустую строку?

з. Что придется сделать в вызывающей функции, чтобы `pr()` можно было использовать так, как показано выше?

6. Предположим, что имеется следующее объявление:

```
char sign = '$';
```

Сколько байтов памяти занимает переменная `sign`? Значение `'$'`? Значение `"$"`?

7. Что выведет следующая программа?

```
#include <stdio.h>
#include <string.h>
#define M1 "How are ya, sweetie? "
char M2[40] = "Beat the clock.";
char * M3 = "chat";
```

```
int main(void)
{
   char words[80];

   printf(M1);
   puts(M1);
   puts(M2);
   puts(M2 + 1);
   strcpy(words,M2);
   strcat(words, " Win a toy.");
   puts(words);
   words[4] = '\0';
   puts(words);
   while (*M3)
      puts(M3++);
   puts(--M3);
   puts(--M3);
   M3 = M1;
   puts(M3);

   return 0;
}
```

8. Что выведет следующая программа?

```
#include <stdio.h>
int main(void)
{
   char str1[] = "gawsie";
   char str2[] = "bletonism";
   char *ps;

   int i = 0;
   for (ps = str1; *ps != '\0'; ps++) {
      if ( *ps == 'a' || *ps == 'e')
            putchar(*ps);
      else
            (*ps)--;
      putchar(*ps);
   }
   putchar('\n');
   while (str2[i] != '\0' ) {
      printf("%c", i % 3 ? str2[i] : '*');
      ++i;
   }

   return 0;
}
```

9. Функцию `s_gets()`, определенную в этой главе, можно переписать в форме с использованием указателей, а не массивов, чтобы исключить из нее переменную `i`. Сделайте это.

10. Функция `strlen()` принимает указатель на строку в качестве аргумента и возвращает длину строки. Напишите свою версию этой функции.

11. Функцию `s_gets()`, определенную в этой главе, можно переписать с использованием функции `strchr()` вместо цикла `while` для обнаружения символа новой строки. Сделайте это.

12. Разработайте функцию, которая принимает указатель на строку в качестве аргумента и возвращает указатель на первый символ пробела, расположенный в указанной позиции внутри строки или после нее. Функция должна возвращать нулевой указатель, если пробелы не найдены.

13. Перепишите программу из листинга 11.21, используя функции из `ctype.h` для того, чтобы программа распознавала правильный ответ независимо от применения пользователем прописных или строчных букв.

Упражнения по программированию

1. Разработайте и протестируйте функцию, которая извлекает из ввода следующие `n` символов (включая символы пробела, табуляции и новой строки), сохраняя результаты в массиве, адрес которого передается в качестве аргумента.

2. Модифицируйте и протестируйте функцию из упражнения 1, обеспечив в ней прекращение ввода после `n` символов либо при достижении первого символа пробела, табуляции или новой строки, в зависимости от того, что произойдет раньше. (Не ограничивайтесь только использованием `scanf()`.)

3. Разработайте и протестируйте функцию, которая читает первое слово из строки ввода в массив и отбрасывает остальную часть строки. Функция должна пропускать ведущие пробельные символы. Определите слово как последовательность символов, не содержащую символов пробела, табуляции или новой строки. Используйте функцию `getchar()`, а не `scanf()`.

4. Разработайте и протестируйте функцию, подобную описанной в упражнении 3, за исключением того, что она принимает второй параметр, указывающий максимальное количество символов, которые могут быть прочитаны.

5. Разработайте и протестируйте функцию, которая ищет в переданной в первом параметре строке первое вхождение символа, заданного во втором параметре. Функция должна возвратить указатель на этот символ, если он найден, и ноль в противном случае. (Поведение этой функции дублирует работу библиотечной функции `strchr()`.) Протестируйте функцию в завершенной программе, которая использует цикл для передачи входных значений созданной функции.

6. Напишите функцию по имени `is_within()`, которая в качестве двух своих параметров принимает символ и указатель на строку. Функция должна возвращать ненулевое значение, если заданный символ содержится в строке, и ноль в противном случае. Протестируйте функцию в завершенной программе, которая использует цикл для передачи входных значений созданной функции.

7. Функция `strncpy(s1,s2,n)` копирует в точности `n` символов из строки `s2` в строку `s1`, при необходимости усекая `s2` или дополняя ее нулевыми символами. Целевая строка может не содержать завершающего нулевого символа, если длина строки `s2` равна или больше `n`. Функция возвращает строку `s1`. Напишите свою версию этой функции и назовите ее `mystrncpy()`. Протестируйте функцию в завершенной программе, которая использует цикл для передачи входных значений созданной функции.

8. Напишите функцию `string_in()`, которая принимает в качестве аргументов два указателя на строки. Если вторая строка содержится внутри первой, функция должна возвратить адрес, с которого начинается вторая строка в первой строке.

Например, вызов `string_in("данные", "ан")` возвратит адрес символа `а` в строке `данные`. В противном случае функция должна возвратить нулевой указатель. Протестируйте функцию в завершенной программе, которая использует цикл для передачи входных значений созданной функции.

9. Напишите функцию, которая заменяет содержимое указанной строки этой же строкой, но с обратным порядком следования символов. Протестируйте функцию в завершенной программе, которая использует цикл для передачи входных значений созданной функции.

10. Напишите функцию, которая принимает строку в качестве аргумента и удаляет из нее все пробелы. Протестируйте эту функцию в программе, которая использует цикл для чтения строк до тех пор, пока не будет введена пустая строка. Программа должна применять эту функцию к каждой входной строке и отображать результат.

11. Напишите программу, которая читает до 10 строк и или до появления `EOF`, в зависимости от того, что произойдет раньше. Функция должна предложить пользователю меню с пятью вариантами: вывод исходного списка строк, вывод строк согласно последовательности сопоставления ASCII, вывод строк в порядке возрастания длины, вывод строк в порядке возрастания длины первого слова в строке и выход из программы. Меню должно отображаться до тех пор, пока пользователь не выберет вариант выхода из программы. Программа должна действительно выполнять запрошенные действия.

12. Напишите программу, которая читает входные данные до тех пор, пока не встретится `EOF`, и выводит количество слов, количество прописных букв, количество строчных букв, количество знаков препинания и количество цифр. Используйте семейство функций `ctype.h`.

13. Напишите программу, которая повторяет на экране аргументы командной строки в обратном порядке. Другими словами, если аргументами командной строки являются `до скорого свидания`, данная программа должна вывести на экран `свидания скорого до`.

14. Напишите программу реализации степенной зависимости, которая работает на основе командной строки. Первым аргументом командной строки должно быть число типа `double`, возводимое в определенную степень, а вторым аргументом – целочисленный показатель степени.

15. Для подготовки реализации функции `atoi()` используйте функции классификации символов. Эта версия должна возвращать значение 0, если строка ввода не является полностью числовой.

16. Напишите программу, которая читает входные данные до тех пор, пока не встретится `EOF`, и выводит их на экран. Программа должна распознавать и реализовывать следующие аргументы командной строки:

    `-p` Вывод входных данных в том виде, как есть.

    `-u` Преобразование входных данных в верхний регистр.

    `-l` Преобразование входных данных в нижний регистр.

    Кроме того, если аргумент входной строки не указан, программа должна вести себя так, как если бы был задан аргумент `-p`.

12

Классы хранения, связывание и управление памятью

В ЭТОЙ **ГЛАВЕ**...

- Ключевые слова: auto, extern, static, register, const,volatile, restricted, _Thread_local, _Atomic
- Функции:rand(),srand(),time(),malloc(), calloc(), free()
- Определение в языке C области видимости переменной (насколько широко она известна) и времени жизни переменной (насколько долго она существует)
- Проектирование более сложных программ

Одна из сильных сторон языка С связана с тем, что он позволяет управлять тонкими аспектами программы. Система управления памятью в С служит иллюстрацией такого управления, позволяя определять, каким функциям известны те или иные переменные и насколько долго переменная существует в программе. Использование хранилища в памяти является еще одним элементом проектного решения, положенного в основу программы.

Классы хранения

Для хранения данных в памяти язык С предлагает пять разных моделей, или *классов хранения*. Чтобы понять доступные варианты, полезно сначала изучить несколько концепций и терминов.

В каждом примере программы, приводимом в этой книге, данные хранятся в памяти. Для этого существует аппаратный аспект – любое сохраненное значение находится в физической памяти. В литературе по С для описания такого участка памяти применяется термин *объект*. Объект может хранить одно или большее количество значений. В определенный момент объект может пока не содержать сохраненного значения, но он будет иметь правильный размер для помещения подходящего значения. (В формулировке *объектно-ориентированное программирование* понятие *объект* используется в более широком смысле для указания объектов классов, определения которых охватывают данные и разрешенные операции на этих данных; С не является языком объектно-ориентированного программирования.)

Имеется также и программный аспект – программе нужен какой-нибудь способ доступа к объекту. Этого можно достичь, например, путем объявления переменной:

```
int entity = 3;
```

Показанное объявление приводит к созданию *идентификатора* по имени `entity`. Идентификатор представляет собой имя, в данном случае такое, которое может применяться для обозначения содержимого отдельного объекта. Идентификаторы следуют соглашениям об именовании переменных, рассмотренным в главе 2. В этом случае идентификатор `entity` отражает способ, которым программное обеспечение (программа на С) указывает объект, хранящийся в аппаратной памяти. Такое объявление также предоставляет значение для сохранения в объекте.

Имя переменной – не единственный метод обозначения объекта. Например, взгляните на следующие объявления:

```
int * pt = &entity;
int ranks[10];
```

В первом случае `pt` представляет собой идентификатор. Он обозначает объект, который содержит адрес. Выражение `*pt` – не идентификатор, поскольку оно не является именем. Тем не менее, оно указывает на объект, в данной ситуации – на тот же самый объект, что и `entity`. В общем случае, как вы можете помнить из главы 3, выражение, которое обозначает объект, называется l-значением. Таким образом, `entity` – это идентификатор, представляющий собой l-значение, а `*pt` – выражение, являющееся l-значением. При тех же объявлениях выражение `ranks + 2 * entity` – не идентификатор (не имя) и не l-значение (не указывает на содержимое ячейки памяти). Но выражение `*(ranks + 2 * entity)` является l-значением, потому что оно указывает на значение определенной ячейки памяти (седьмого элемента массива `ranks`). Кстати говоря, объявление `ranks` приводит к созданию объекта, способного хранить 10 значений `int`, и каждый элемент массива также представляет собой объект.

Если, как и во всех этих примерах, l-значение можно использовать для изменения значения внутри объекта, мы имеем дело с *модифицируемым l-значением*. Теперь рассмотрим следующее объявление:

```
const char * pc = "Это строковый литерал!";
```

Оно приводит к тому, что программа сохраняет в памяти содержимое строкового литерала, и этот массив символьных значений является объектом. Каждый символ в массиве также представляет собой объект, т.к. к нему можно обращаться индивидуально. Объявление также создает объект, который имеет идентификатор pc и хранит адрес данной строки. Идентификатор pc – это модифицируемое l-значение, поскольку его можно переустанавливать для указания на другие строки. Ключевое слово const предотвращает изменение содержимого строки, на которую указывает pc, но не изменение того, на какую строку он указывает. Таким образом, выражение *pc, обозначающее объект данных с символом 'Э', является l-значением, но не модифицируемым l-значением. Аналогично, сам строковый литерал, указывая на объект, который содержит символьную строку, представляет собой l-значение, не допускающее модификации.

Объект можно описать в терминах его *продолжительности хранения*, которая указывает, насколько долго он остается в памяти. Идентификатор, применяемый для доступа к этому объекту, может быть описан посредством его *области видимости* и *связывания*, которые вместе указывают, в какой части программы этот идентификатор разрешено использовать. Разные классы хранения предлагают различные сочетания области видимости, связывания и продолжительности хранения. Допускается существование идентификаторов, совместно используемых в нескольких файлах исходного кода, идентификаторов, которые могут применяться в любых функциях внутри одного файла, идентификаторов, используемых только внутри отдельной функции, и даже идентификаторов, применяемых лишь в каком-то разделе функции. Одни объекты могут существовать на протяжении времени жизни целой программы, а другие – только во время выполнения функции, которая их содержит. В параллельном программировании можно иметь объекты, которые существуют в течение выполнения отдельного потока. Можно также хранить данные в памяти, которая явно выделяется и освобождается посредством вызовов функций. Давайте посмотрим, что означают термины *область видимости*, *связывание* и *продолжительность хранения*. После этого приступим к изучению конкретных классов хранения.

Область видимости

Область видимости описывает участок или участки программы, где можно обращаться к идентификатору. Переменная в С имеет одну из следующих областей видимости: в пределах блока, в пределах функции, в пределах прототипа функции и в пределах файла. В рассмотренных до сих пор примерах программ для переменных использовалась в основном область видимости на уровне блока. Как вы помните, *блок* – это часть кода, содержащаяся между открывающей фигурной скобкой и соответствующей ей закрывающей скобкой. Например, блоком является все тело функции. Любой составной оператор внутри функции также считается блоком. Переменная, определенная в блоке, имеет *область видимости в пределах блока*, и она видна от места, где она определена, до конца блока, содержащего определение. Кроме того, формальные параметры функции, хотя они появляются до открывающей фигурной скобки функции, имеют область видимости в пределах блока и принадлежат блоку, содержащему тело функции. Таким образом, все локальные переменные, которые применялись до сих пор, включая формальные параметры функций, располагали областью видимости в пределах блока. Следовательно, переменные cleo и patrick в приведенном ниже

коде имеют область видимости в пределах блока, простирающегося до закрывающей фигурной скобки:

```
double blocky(double cleo)
{
   double patrick = 0.0;
   ...
   return patrick;
}
```

Переменные, объявленные во внутреннем блоке, получают область видимости, ограниченную только этим блоком:

```
double blocky(double cleo)
{
   double patrick = 0.0;
   int i;
   for (i = 0; i < 10; i++)
   {
      double q = cleo * i;     // начало области видимости для q
      ...
      patrick *= q;
   }                           // конец области видимости для q
   ...
   return patrick;
}
```

В этом примере область видимости q ограничена внутренним блоком, и доступ к q может иметь только код внутри этого блока.

По традиции переменные с областью видимости в пределах блока должны объявляться в начале блока. В стандарте C99 это требование было ослаблено, и переменные разрешено объявлять в любом месте блока. Одна из новых возможностей связана с объявлением внутри управляющего раздела цикла for. То есть теперь можно поступать так:

```
for (int i = 0; i < 10; i++)
   printf("Возможность C99: i = %d", i);
```

Как часть этой новой возможности, стандарт C99 расширил концепцию блока путем включения в нее кода, управляющего циклами for, while, do while или оператором if, даже если фигурные скобки при этом не используются. Таким образом, в предыдущем цикле for переменная i считается частью блока цикла for. Следовательно, ее область видимости ограничена циклом for. После того, как выполнения покинет цикл for, эта переменная i больше не видна программе.

Область видимости в пределах функции применяется только к меткам, применяемым с операторами goto. Это означает, что если метка впервые появляется во внутреннем блоке функции, ее область видимости простирается на всю функцию. Если бы можно было использовать одну и ту же метку внутри двух отдельных блоков, возникла бы путаница, поэтому область видимости в пределах функции для меток предотвращает такую ситуацию.

Область видимости в пределах прототипа функции применяется к именам переменных, используемым в прототипах функций, как в следующем случае:

```
int mighty(int mouse, double large);
```

Область видимости в пределах прототипа функции распространяется от места определения переменной до конца объявления прототипа. Это значит, что при обработ-

ке аргумента прототипа функции компилятор интересует только тип аргумента. Если указаны имена, то обычно они не играют никакой роли и не обязательно должны совпадать с именами, которые применяются в определении функции. Имена играют небольшую роль в случае параметров, имеющих типы массивов переменной длины:

```
void use_a_VLA(int n, int m, ar[n][m]);
```

При использовании имен в скобках это должны быть имена, объявленные ранее в прототипе.

Переменная, определение которой находится за рамками любой функции, имеет *область видимости в пределах файла*. Переменная, располагающая областью видимости в пределах файла, видна от места ее определения и до конца файла, который содержит ее определение. Взгляните на показанный ниже пример:

```
#include <stdio.h>
int units = 0;         /* переменная с областью видимости в пределах файла */
void critic(void);
int main(void)
{

}
void critic(void)
{

}
```

Здесь переменная `units` имеет область видимости в пределах файла и может применяться и в `main()`, и в `critic()`.(Точнее, `units` имеет область видимости в пределах файла и внешнее связывание; отличие мы раскроем в следующем разделе.) Поскольку переменные с областью видимости в пределах файла могут использоваться в более чем одной функции, они еще называются *глобальными переменными*.

НА ЗАМЕТКУ! Единицы и файлы трансляции

То, что вы видите как несколько файлов, для компилятора может выглядеть как единственный файл. Предположим для примера, и это случается довольно часто, что вы включаете один или больше заголовочных файлов (с расширением `.h`) в файл исходного кода (с расширением `.c`). В свою очередь, заголовочный файл может включать другие заголовочные файлы. В итоге могут быть задействованы многие физические файлы. Однако предпроцессор С по существу заменяет директиву `#include` содержимым заголовочного файла. Таким образом, компилятор видит единственный файл, содержащий информацию из вашего файла исходного кода и всех заголовочных файлов. Такой файл называется *единицей трансляции*. Когда мы описываем переменную как имеющую область видимости в пределах файла, на самом деле она будет видимой целой единице трансляции. Если ваша программа состоит из нескольких файлов исходного кода, тогда она будет насчитывать и несколько единиц трансляции, каждая из которых соответствует файлу исходного кода и включаемым в него файлам.

Связывание

Давайте поговорим о связывании. Переменная в С имеет одно из следующих связываний: внешнее связывание, внутреннее связывание или отсутствие связывания. Переменные с областью видимости в пределах блока, функции или прототипа функции не имеют связывания. Это означает, что они являются закрытыми для блока, функции или прототипа, в котором определены. Переменная с областью видимости в пределах файла может иметь либо внутреннее, либо внешнее связывание. Переменная

с внешним связыванием может применяться в любом месте многофайловой программы, а переменная с внутренним связыванием — где угодно в единице трансляции.

НА ЗАМЕТКУ! Формальные и неформальные термины

В стандарте С для описания области видимости, ограниченной одной единицей трансляции (файл исходного кода плюс его включаемые заголовочные файлы), используется формулировка "область видимости в пределах файла с внутренним связыванием", а для описания области видимости, которая (во всяком случае, потенциально) распространяется на другие единицы трансляции — формулировка "область видимости в пределах файла с внешним связыванием". Но у программистов не всегда есть время и терпение применять такие термины. Распространение получили сокращения "область видимости в пределах файла" для "области видимости в пределах файла с внутренним связыванием" и "глобальная область видимости" или "область видимости в пределах программы" для "области видимости в пределах файла с внешним связыванием".

Как же тогда выяснить, внутреннее или внешнее связывание имеет переменная с областью видимости в пределах файла? Вы должны посмотреть, используется ли во внешнем определении спецификатор класса хранения `static`:

```
int giants = 5;          // область видимости в пределах файла,
                         // внешнее связывание
static int dodgers = 3;  // область видимости в пределах файла,
                         // внутреннее связывание
int main()
{

}
...
```

Переменная `giants` может применяться в других файлах, которые представляют собой составные части той же самой программы. Переменная `dodgers` является закрытой для данного конкретного файла, но может использоваться любой функцией в этом файле.

Продолжительность хранения

Область видимости и связывание описывают видимость идентификаторов. Продолжительность хранения характеризует постоянство объектов, доступных через эти идентификаторы. Объект в С имеет одну из следующих четырех продолжительностей хранения: статическую, потоковую, автоматическую или выделенную.

Если объект имеет статическую продолжительность хранения, он существует на протяжении времени выполнения программы. Переменные с областью видимости в пределах файла имеют статическую продолжительность хранения. Обратите внимание, что для переменных с областью видимости в пределах файла ключевое слово `static` указывает тип связывания, а не продолжительность хранения. Переменная с областью видимости в пределах файла, объявленная с применением `static`, имеет внутреннее связывание, но все переменные с областью видимости в пределах файла, имеющие внутреннее или внешнее связывание, обладают статической продолжительностью хранения.

Потоковая продолжительность хранения вступает в игру при параллельном программировании, когда выполнение программы может быть разделено на несколько потоков. Объект с потоковой продолжительностью хранения существует с момента его объявления и до завершения потока. Такой объект создается, когда объявление, которое иначе привело бы к созданию объекта с областью видимости в пределах файла,

модифицировано с помощью ключевого слова _Thread_local. Когда переменная объявлена с таким спецификатором, каждый поток получает собственную закрытую копию этой переменной.

Переменные с областью видимости в пределах блока обычно имеют автоматическую продолжительность хранения. Память для этих переменных выделяется, когда поток управления входит в блок, где они определены, и освобождается, когда поток управления покидает этот блок. Идея заключается в том, что память, используемая для автоматических переменных, является рабочим пространством или временной памятью, которая может применяться многократно. Например, после завершения вызова функции память, которую функция использовала для своих переменных, может быть задействована при вызове следующей функции.

Массивы переменной длины демонстрируют небольшое исключение в том, что они существуют от места своего объявления и до конца блока, а не от начала блока и до его конца.

Локальные переменные, которые мы применяли до сих пор, попадают в категорию автоматической продолжительности хранения. Например, в следующем коде переменные number и index появляются каждый раз, когда функция bore() вызывается, и исчезают после ее завершения:

```
void bore(int number)
{
   int index;
   for (index = 0; index < number; index++)
      puts("Выполнение привычной работы.\n");
   return 0;
}
```

Тем не менее, переменная может иметь область видимости в пределах блока, но статическую продолжительность хранения. Чтобы создать такую переменную, объявите ее внутри блока и добавьте в объявление ключевое слово static:

```
void more(int number)
{
   int index;
   static int ct = 0;
   ...
   return 0;
}
```

Здесь переменная ct хранится в статической памяти; она существует с момента загрузки программы в память и вплоть до завершения выполнения программы. Но область видимости ct ограничена блоком функции more(). Только во время выполнения этой функции программа может использовать переменную ct для доступа к объекту, который она обозначает. (Тем не менее, можно разрешить непрямой доступ, позволив функции предоставить адрес хранилища другим функциям, например, с помощью параметра типа указателя или возвращаемого значения.)

Область видимости, связывание и продолжительность хранения используются в С в целях определения нескольких схем хранения для переменных. В этой книге не рассматривается параллельное программирование, поэтому мы не будем касаться данного аспекта. Выделенная продолжительность хранения будет обсуждаться позже в главе. В итоге остаются пять классов хранения: автоматический, регистровый, статический с областью видимости в пределах блока, статический с внешним связыванием и статический с внутренним связыванием. Комбинации представлены в табл. 12.1.

Таблица 12.1. Пять классов хранения

Класс хранения	Продолжительность хранения	Область видимости	Связывание	Объявление
Автоматический	Автоматическая	В пределах блока	Нет	В блоке
Регистровый	Автоматическая	В пределах блока	Нет	В блоке с указанием ключевого слова `register`
Статический с внешним связыванием	Статическая	В пределах файла	Внешнее	За рамками всех функций
Статический с внутренним связыванием	Статическая	В пределах файла	Внутреннее	За рамками всех функций с указанием ключевого слова `static`
Статический без связывания	Статическая	В пределах файла	Нет	В блоке с указанием ключевого слова `static`

Теперь, когда мы раскрыли понятия области хранения, связывания и продолжительности хранения, можно переходить к более детальным исследованиям этих классов хранения.

Автоматические переменные

Переменная, принадлежащая к автоматическому классу хранения, имеет автоматическую продолжительность хранения, область видимости в пределах блока и не имеет связывания. По умолчанию любая переменная, объявленная в блоке или в заголовке функции, относится к автоматическому классу хранения. Однако вы можете совершенно ясно сформулировать свои намерения, явным образом указав ключевое слово `auto`:

```
int main(void)
{
   auto int plox;
```

Это можно делать, скажем, для документирования того факта, что вы намеренно переопределяете внешнюю переменную, или для подчеркивания важности того, что класс хранения переменной не должен изменяться. Ключевое слово `auto` называется *спецификатором класса хранения*. В C++ ключевое слово `auto` предназначено для совершенно другой цели, поэтому просто не применяйте `auto` в качестве спецификатора класса хранения, чтобы добиться большей совместимости между С и C++.

Область видимости в пределах блока и отсутствие связывания подразумевает, что доступ к этой переменной по имени может осуществляться только в блоке, где переменная определена. (Конечно, посредством аргументов значение и адрес переменной можно передать в другую функцию, но это будут уже косвенные сведения.) В другой функции может использоваться переменная с тем же самым именем, но это будет независимая переменная, хранящаяся в другой ячейке памяти.

Как вы, возможно, помните, автоматическая продолжительность хранения означает, что переменная начинает свое существование, когда поток управления входит в блок, который содержит объявление этой переменной. После того как поток управления покинет блок, автоматическая переменная исчезнет. Ячейка памяти, которую она занимала, может применяться для чего-то другого, хотя и необязательно.

Давайте более внимательно посмотрим на вложенные блоки. Переменная известна только в блоке, в котором она объявлена, и в любых блоках, размещенных внутри этого блока:

```
int loop(int n)
{
    int m;          // m находится в области видимости
    scanf("%d", &m);
    {
        int i;      // и m, и i находятся в области видимости
        for (i = m; i < n; i++)
            puts("i is local to a sub-block\n");
    }
    return m;       // m в области видимости, i исчезла
}
```

В данном коде переменная i является видимой только во внутренних скобках. Если вы попытаетесь воспользоваться этой переменной до или после внутреннего блока, компилятор сообщит об ошибке. Обычно такой прием при проектировании программ не применяется. Тем не менее, временами удобно определять переменную в подблоке, если она не используется в других местах. В этом случае можно документировать назначение переменной близко к месту ее применения.

Кроме того, переменная не будет оставаться неиспользуемой, попусту занимая место, когда она больше не нужна. Так как переменные n и m определены в заголовке функции и во внешнем блоке, они находятся в области видимости всей функции и существуют вплоть до ее завершения.

Что, если вы объявите во внутреннем блоке переменную, которая имеет такое же имя, как переменная во внешнем блоке? Тогда имя, определенное внутри блока, соответствует переменной, которая применяется в этом блоке. Мы говорим, что имя *скрывает* внешнее определение. Однако когда поток управления покидает внутренний блок, внешняя переменная возвращается в область видимости. Эти и другие аспекты проиллюстрированы в листинге 12.1.

Листинг 12.1. Программа hiding.c

```
// hiding.c -- переменные в блоках
#include <stdio.h>
int main()
{
   int x = 30;              // исходная переменная x

   printf("x во внешнем блоке: %d по адресу %p\n", x, &x);
   {
      int x = 77;           // новая переменная x, скрывающая первую x
      printf("x во внутреннем блоке: %d по адресу %p\n", x, &x);
   }
   printf("x во внешнем блоке: %d по адресу %p\n", x, &x);
   while (x++ < 33)         // исходная переменная x
   {
      int x = 100;          // новая переменная x, скрывающая первую x
      x++;
      printf("x в цикле while: %d по адресу %p\n", x, &x);
   }
   printf("x во внешнем блоке: %d по адресу %p\n", x, &x);

   return 0;
}
```

Вот результаты пробного запуска:

```
x во внешнем блоке: 30 по адресу 0x7fff5fbff8c8
x во внутреннем блоке: 77 по адресу 0x7fff5fbff8c4
x во внешнем блоке: 30 по адресу 0x7fff5fbff8c8
x в цикле while: 101 по адресу 0x7fff5fbff8c0
x в цикле while: 101 по адресу 0x7fff5fbff8c0
x в цикле while: 101 по адресу 0x7fff5fbff8c0
x во внешнем блоке: 34 по адресу 0x7fff5fbff8c8
```

Первым делом программа создает переменную x со значением 30, как показывает первый оператор printf(). Затем она определяет новую переменную x со значением 77, о чем сообщает второй оператор printf(). Это новая переменная, скрывающая первую переменную x, значение и адрес которой снова выводятся третьим оператором printf(). Данный оператор находится после первого внутреннего блока и отображает первоначальное значение x, демонстрируя тем самым, что эта переменная никуда не исчезала и не изменялась.

Вероятно, наиболее интригующей частью программы является цикл while. В условии проверки цикла while задействована исходная переменная x:

```
while(x++ < 33)
```

Однако внутри цикла программа видит третью переменную x, т.е. ту, которая определена в рамках блока цикла while. Таким образом, когда в теле цикла используется выражение x++, в нем участвует новая переменная x, значение которой инкрементируется до 101 и затем отображается. По завершении каждой итерации цикла эта новая переменная x исчезает.

Далее в условии проверки цикла применяется и инкрементируется исходная переменная x, снова происходит вход в блок цикла, и опять создается новая переменная x. В этом примере переменная x создается и уничтожается три раза.

Обратите внимание, что для прекращения выполнения цикл должен инкрементировать x в условии проверки, т.к. инкрементирование x в теле цикла приводит к увеличению значения другой переменной x, а не той, которая задействована в условии проверки.

Хотя применяемый конкретный компилятор не использует повторно ячейку памяти переменной x внутреннего блока для версии x из цикла while, некоторые компиляторы делают это.

Назначение этого примера вовсе не в том, чтобы поощрять написание кода в таком стиле. Он служит лишь иллюстрацией того, что происходит, когда вы определяете переменные внутри блока. (Учитывая многообразие имен, доступных благодаря правилам именования C, выбор имени, отличающегося от x, не должен вызывать особые затруднения.)

Блоки без фигурных скобок

Упомянутая ранее возможность стандарта C99 заключается в том, что операторы, которые являются частью цикла или оператора if, квалифицируются как блок, даже если фигурные скобки ({ }) не указаны. Выражаясь более точно, полный цикл — это подблок содержащего его блока, а тело цикла — подблок блока полного цикла. Аналогично, оператор if представляет собой блок, а связанный с ним оператор — подблок оператора if. Описанные правила влияют на то, где вы можете объявлять переменную, и на область видимости этой переменной.

В листинге 12.2 показано, как это работает в цикле for.

Листинг 12.2. Программа forc99.c

```
// forc99.c -- новые правила для блоков в C99
#include <stdio.h>
int main()
{
   int n = 8;

   printf("  Первоначально n = %d по адресу %p\n", n, &n);
   for (int n = 1; n < 3; n++)
      printf("        цикл 1: n = %d по адресу %p\n", n, &n);
   printf("  После цикла 1 n = %d по адресу %p\n", n, &n);
   for (int n = 1; n < 3; n++)
   {
      printf(" индекс цикла 2 n = %d по адресу %p\n", n, &n);
      int n = 6;
      printf("        цикл 2: n = %d по адресу %p\n", n, &n);
      n++;
   }
   printf("  После цикла 2 n = %d по адресу %p\n", n, &n);

   return 0;
}
```

Ниже приведен вывод, исходя из предположения, что компилятор поддерживает современные возможности C:

```
  Первоначально n = 8 по адресу 0x7fff5fbff8c8
         цикл 1: n = 1 по адресу 0x7fff5fbff8c4
         цикл 1: n = 2 по адресу 0x7fff5fbff8c4
  После цикла 1 n = 8 по адресу 0x7fff5fbff8c8
 индекс цикла 2 n = 1 по адресу 0x7fff5fbff8c0
         цикл 2: n = 6 по адресу 0x7fff5fbff8bc
 индекс цикла 2 n = 2 по адресу 0x7fff5fbff8c0
         цикл 2: n = 6 по адресу 0x7fff5fbff8bc
  После цикла 2 n = 8 по адресу 0x7fff5fbff8c8
```

НА ЗАМЕТКУ! Поддержка C99 и C11

Некоторые компиляторы могут не поддерживать правила области видимости C99/C11. (В настоящее время одним из таких компиляторов является Microsoft Visual Studio 2012.) Другие компиляторы могут предлагать опцию включения этих правил. Например, на момент написания данной книги компилятор GCC по умолчанию поддерживает многие возможности C99, но для активизации средств, применяемых в листинге 12.2, требует указания опции `-std=c99`:

```
gcc -std=c99 forc99.c
```

Подобным же образом, версии GCC и Clang могут требовать использования опции `-std=c1x` или `-std=c11` для распознавания средств C11.

Переменная `n`, объявленная в управляющем разделе первого цикла `for`, имеет область видимости до конца цикла и скрывает исходную переменную `n`. Но после того как управление покидает цикл, исходная переменная `n` возвращается в область видимости.

Во втором цикле `for` переменная `n`, объявленная как индекс цикла, скрывает исходную переменную `n`. Затем переменная `n`, объявленная внутри тела цикла, скрывает индекс цикла `n`. Как только программа завершит выполнение тела, переменная `n`, объявленная в теле, исчезает, а в проверке цикла участвует индекс `n`. Когда завершится выполнения всего цикла, в области видимости появляется исходная переменная `n`.

И снова отметим, что нет никакой нужды многократно применять одного и того же имени для переменной, но вы должны знать, что произойдет, если вы все-таки решите поступить так.

Инициализация автоматических переменных

Автоматические переменные не инициализируются до тех пор, пока вы не сделаете это явно. Взгляните на следующие объявления:

```
int main(void)
{
    int repid;
    int tents = 5;
```

Переменная tents инициализируется значением 5, но repid получает значение, которое раньше находилось в области памяти, выделенной под эту переменную. Нельзя рассчитывать на то, что этим значением будет 0. Вы можете инициализировать автоматическую переменную неконстантным выражением при условии, что все задействованные в нем переменные были определены раньше:

```
int main(void)
{
    int ruth = 1;
    int rance = 5 * ruth;    // используется ранее определенная переменная
```

Регистровые переменные

Переменные обычно хранятся в памяти компьютера. При благоприятном стечении обстоятельств регистровые переменные хранятся в регистрах центрального процессора, или, в общем случае, в самой быстрой имеющейся памяти, что обеспечивает доступ и манипулирование ими с меньшими затратами времени, чем для рядовых переменных. Поскольку регистровая переменная может находиться в регистре, а не в памяти, получить адрес такой переменной не удастся. В большинстве других отношений регистровые переменные ничем не отличаются от автоматических переменных. То есть они имеют область видимости в пределах блока, не имеют связывания и имеют автоматическую продолжительность хранения. Переменная объявляется с использованием спецификатора класса хранения register:

```
int main(void)
{
    register int quick;
```

Мы говорим "при благоприятном стечении обстоятельств", потому что объявление переменной как регистровой является скорее запросом, чем прямым указанием. Компилятор должен сопоставить ваши требования с количеством доступных регистров или объема быстродействующей памяти, или же он может просто проигнорировать запрос, и ваше пожелание не будет удовлетворено. В таком случае переменная становится обычной автоматической переменной; тем не менее, применять к ней операцию взятия адреса по-прежнему нельзя. Вы можете запросить, чтобы формальные параметры были регистровыми переменными. Для этого просто воспользуйтесь ключевым словом register в заголовке функции:

```
void macho(register int n)
```

Типы, которые допускается объявлять как register, могут оказаться ограниченными. Например, регистры в процессоре могут быть недостаточно большими, чтобы умещать тип double.

Статические переменные с областью видимости в пределах блока

Название *статическая переменная* звучит как взаимоисключающее, вроде переменной, которая не может быть изменена. В действительности характеристика *статическая* означает, что переменная остается помещенной в память, а не обязательно относится к значению. Переменные с областью видимостью в пределах файла автоматически (и обязательно) имеют статическую продолжительность хранения. Как упоминалось ранее, можно также создавать локальные переменные, имеющие область видимости в пределах блока, но статистическую продолжительность хранения. Такие переменные обладают такой же областью видимости, как автоматические переменные, однако они не исчезают, когда содержащая их функция завершает свою работу. Другими словами, такие переменные имеют область видимости в пределах блока, не имеют связывания, но имеют статическую продолжительность хранения. Компьютер помнит их значения от одного вызова функции до следующего. Такие переменные создаются в результате объявления в блоке (что обеспечивает область видимости в пределах блока и отсутствие связывания) со спецификатором класса хранения `static` (предоставляющим статическую продолжительность хранения).

Пример в листинге 12.3 служит иллюстрацией этого приема.

Листинг 12.3. Программа `loc_stat.c`

```
/* loc_stat.c -- использование локальной статической переменной */
#include <stdio.h>
void trystat(void);

int main(void)
{
   int count;

   for (count = 1; count <= 3; count++)
   {
      printf("Начинается итерация %d:\n", count);
      trystat();
   }

   return 0;
}

void trystat(void)
{
   int fade = 1;
   static int stay = 1;

   printf("fade = %d и stay = %d\n", fade++, stay++);
}
```

Обратите внимание, что `trystat()` инкрементирует каждую переменную после вывода ее значения. Запуск программы дает следующий вывод:

```
Начинается итерация 1:
fade = 1 и stay = 1
Начинается итерация 2:
fade = 1 и stay = 2
Начинается итерация 3:
fade = 1 и stay = 3
```

Статическая переменная stay запоминает, что ее значение было увеличено на 1, но переменная fade каждый раз начинается заново. Это отражает отличие в инициализации: fade инициализируется при каждом вызове trystat(), а stay — только один раз, когда функция trystat() компилируется. Статические переменные инициализируются нулем, если они были явно инициализированы другим значением.

Два следующих объявления выглядят похожими:

```
int fade = 1;
static int stay = 1;
```

Тем не менее, первый оператор в действительности является частью функции trystat() и выполняется каждый раз, когда функция вызвана. Это действие времени выполнения. Второй оператор на самом деле не относится к функции trystat(). Если вы примените отладчик для пошагового выполнения программы, то увидите, что программа как бы пропускает этот шаг. Причина в том, что после того, как программа загрузилась в память, статические переменные и внешние переменные уже находятся в нужных местах. Помещение оператора объявления в функцию trystat() сообщает компилятору, что только функции trystat() разрешено видеть данную переменную; это не оператор, который исполняется во время выполнения.

Использовать модификатор static для параметров функции нельзя:

```
int wontwork(static int flu);   // не разрешено
```

Другим термином для статической переменной с областью видимости в пределах блока является "локальная статическая переменная". Кроме того, если вы читали раннюю литературу по С, то обнаружите, что этот класс хранения называли *внутренним статическим классом хранения*. Тем не менее, слово *внутренний* применялось для указания на объявление внутри функции, а не на внутреннее связывание.

Статические переменные с внешним связыванием

Статическая переменная с внешним связыванием имеет область видимости в пределах файла, внешнее связывание и статическую продолжительность хранения. Такой класс иногда называют *внешним классом хранения*, а переменные этого типа — *внешними переменными*. Внешняя переменная создается путем размещения определяющего объявления за рамками всех функций. Согласно документации, внешняя переменная может дополнительно быть объявлена внутри функции, в которой она используется, с применением ключевого слова extern. Если какая-то внешняя переменная определена в одном файле исходного кода и используется во втором файле исходного кода, то объявление этой переменной во втором файле с ключевым словом extern является обязательным. Объявления выглядят следующим образом:

```
int Errupt;              /* внешне определенная переменная   */
double Up[100];          /* внешне определенный массив       */
extern char Coal;        /* обязательное объявление, если    */
                         /* Coal определяется в другом файле */
void next(void);
int main(void)
{
   extern int Errupt;    /* необязательное объявление       */
   extern double Up[];   /* необязательное объявление       */
   ...
}
void next(void)
{
}
```

Обратите внимание, что вы не обязаны указывать размерность массива в необязательном объявлении double Up. Это объясняется тем, что исходное объявление уже предоставило такую информацию. Группу объявлений extern внутри main() можно полностью опустить, т.к. внешние объявления имеют область видимости в пределах файла, поэтому они известны от места объявления и до конца файла. Однако они служат для документирования намерений применять эти переменные в main().

Если ключевое слово extern отсутствует в объявлении внутри функции, создается отдельная автоматическая переменная. То есть, замена

```
extern int Errupt;
```

объявлением

```
int Errupt;
```

в main() приводит к тому, что компилятор создает автоматическую переменную по имени Errupt — отдельную локальную переменную, которая отличается от исходной переменной Errupt. Эта локальная переменная будет находиться в области видимости во время выполнения main(), но для других функций, таких как next(), расположенных в том же самом файле, в области видимости будет внешняя переменная Errupt. Короче говоря, переменная с областью видимости в пределах блока "скрывает" переменную с тем же самым именем, имеющую область видимости в пределах файла, когда происходит выполнение операторов в этом блоке. Если по какой-то маловероятной причине вам действительно необходима локальная переменная, имеющая то же имя, что и глобальная переменная, можете воспользоваться в локальном объявлении спецификатором класса хранения auto, чтобы явно документировать свой выбор.

Внешние переменные имеют статическую продолжительность хранения. Таким образом, массив Up существует и сохраняет свои значения независимо от того, выполняется main(), next() или какая-то другая функция.

В следующих трех примерах показаны четыре возможных комбинации внешних и автоматических переменных. В примере 1 присутствует одна внешняя переменная Hocus, которая известна и main(), и magic().

```
/* Пример 1 */
int Hocus;
int magic();
int main(void)
{
   extern int Hocus;  // переменная Hocus объявлена как внешняя
   ...
}
int magic()
{
   extern int Hocus;  // та же переменная Hocus, что и выше
   ...
}
```

В примере 2 имеется одна внешняя переменная Hocus, известная обеим функциям. На этот раз функция magic() знает о ней по умолчанию.

```
/* Пример 2 */
int Hocus;
int magic();
int main(void)
{
   extern int Hocus;  // переменная Hocus объявлена как внешняя
   ...
}
```

```
int magic()
{
                    // переменная Hocus не объявлена, но известна
   
}
```

В примере 3 создаются четыре переменных. Переменная Hocus в main() является автоматической по умолчанию и локальной для main(). Переменная Hocus в magic() явно объявлена как автоматическая и известна только magic(). Внешняя переменная Hocus не известна main() или magic(), но будет известна любой другой функции в данном файле, не имеющей собственной локальной переменной Hocus. Наконец, Pocus — это внешняя переменная, которая известна magic(), но не main(), потому что Pocus находится за main().

```
/* Пример 3 */
int Hocus;
int magic();
int main(void)
{
  int Hocus; // переменная Hocus объявлена, по умолчанию является автоматической
    ...
}
int Pocus;
int magic()
{
  auto int Hocus;  // локальная переменная Hocus объявлена как автоматическая
    ...
```

Приведенные примеры иллюстрируют область видимости внешних переменных, которая простирается от места их объявления и до конца файла. Они также отражают время жизни переменных. Внешние переменные Hocus и Pocus существуют на протяжении всего времени выполнения программы, а поскольку они не ограничены какой-то одной функцией, они не исчезают после завершения конкретной функции.

Инициализация внешних переменных

Как и автоматические, внешние переменные могут инициализироваться явно. В отличие от автоматических, внешние переменные по умолчанию инициализируются нулем, если вы не инициализировали их. Это правило применимо также к элементам внешне определенного массива. Однако для инициализации переменных с областью видимости в пределах файла можно использовать только константные выражения, что отличается от случая автоматических переменных.

```
int x = 10;               // допустимо, 10 — это константа
int y = 3 + 20;           // допустимо, константное выражение
size_t z = sizeof(int);   // допустимо, константное выражение
int x2 = 2 * x;           // недопустимо, x — это переменная
```

(При условии, что типом не является массив, выражение sizeof считается константным.)

Использование внешней переменной

Давайте рассмотрим простой пример, в котором задействована внешняя переменная. В частности, предположим, что две функции с именами main() и critic() должны иметь доступ к переменной units. Это можно сделать, объявив units за пределами ранее упомянутых двух функций, как показано в листинге 12.4. (Примечание: назначение данного примера — демонстрация работы внешней переменной, а не ее типичное применение.)

Листинг 12.4. Программа global.c

```
/* global.c -- использование внешней переменной */
#include <stdio.h>
int units = 0;         /* внешняя переменная */
void critic(void);
int main(void)
{
   extern int units;  /* необязательное повторное объявление */

   printf("Сколько фунтов весит маленький бочонок масла?\n");
   scanf("%d", &units);
   while (units != 56)
      critic();
   printf("Вы знали это!\n");

   return 0;
}

void critic(void)
{
   /* необязательное повторное объявление опущено */
   printf("Вам не повезло. Попробуйте еще раз.\n");
   scanf("%d", &units);
}
```

Вот результаты пробного запуска:

```
Сколько фунтов весит маленький бочонок масла?
14
Вам не повезло. Попробуйте еще раз.
56
Вы знали это!
```

Обратите внимание, что второе значение для `units` читается функцией `critic()`, но `main()` также известно новое значение после завершения цикла `while`. Таким образом, и `main()`, и `critic()` используют идентификатор `units` для доступа к одной и той же переменной. В рамках терминологии С мы говорим, что переменная `units` имеет область видимости в пределах файла, внешнее связывание и статическую продолжительность хранения.

Мы сделали `units` внешней переменной, определив ее за пределами определений всех функций. Это и все, что необходимо сделать для обеспечения доступности `units` всем последующим функциям в файле.

Давайте посмотрим на некоторые детали. Прежде всего, объявление переменной `units` там, где оно находится, делает ее доступной объявленным далее функциям без дополнительных усилий. Следовательно, функция `critics()` пользуется переменной `units`.

Аналогично, ничего не придется предпринимать для предоставления доступа к `units` функции `main()`. Однако в `main()` имеется такое объявление:

```
extern int units;
```

В рассматриваемом примере это объявление является главным образом формой документирования. Спецификатор класса хранения `extern` сообщает компилятору, что любое упоминание `units` в данной функции относится к переменной, объявленной за пределами этой функции, возможно, даже вне самого файла. И снова `main()` и `critic()` работают с внешне определенной переменной `units`.

Внешние имена

Стандарты C99 и C11 требуют, чтобы компиляторы распознавали первые 63 символа для локальных идентификаторов и первые 31 символ для внешних идентификаторов. Это корректирует предыдущее требование по распознаванию первых 31 символа для локальных и первых 6 символов для внешних идентификаторов. Вполне возможно, что вы имеете дело со старыми правилами. Причина того, что правила для имен внешних переменных являются более ограничивающими, чем правила для имен локальных переменных, связана с тем, что внешние имена должны подчиняться правилам локальной среды, которые могут быть более жесткими.

Определения и объявления

А теперь давайте внимательнее посмотрим на отличие между определением переменной и ее объявлением. Взгляните на следующий пример:

```
int tern = 1;        /* переменная tern определена                    */
main()
{
  external int tern; /*использование tern, определенной где-то в другом месте*/
```

Здесь переменная tern объявлена дважды. Первое объявление приводит к тому, что для переменной отводится место в памяти. Оно образует определение переменной. Второе объявление просто указывает компилятору на необходимость применения переменной tern, которая была создана ранее, так что это не определение. Первое объявление называется *определяющим объявлением*, а второе — *ссылочным объявлением*. Ключевое слово extern говорит о том, что объявление не является определением, т.к. оно инструктирует компилятор искать определение где-то в другом месте.

Предположим, что вы записали следующий код:

```
extern int tern;
int main(void)
{
```

Компилятор предположит, что действительное определение tern находится в другом месте программы, возможно, в другом файле. Это объявление не приводит к выделению пространства в памяти. Таким образом, не используйте ключевое слово extern для создания внешнего определения; применяйте его только для *ссылки* на существующее внешнее определение.

Внешняя переменная может быть инициализирована только один раз, и это должно делаться при определении переменной. Взгляните на следующий код:

```
// файл one.c
char permis = 'N';
...
// файл two.c
extern char permis = 'Y';  /* ошибка */
```

Ошибка заключается в том, что определяющее объявление в файле one.c уже было создано, и оно инициализировало переменную permis.

Статические переменные с внутренним связыванием

Переменные с этим классом хранения имеют статическую продолжительность хранения, область видимости в пределах файла и внутреннее связывание. Такая переменная создается путем ее определения вне любых функций (как и в случае внешней переменной) с указанием спецификатора класса хранения static:

```
static int svil = 1;  // статическая переменная, внутреннее связывание
int main(void)
{
```

Переменные подобного рода когда-то получили название *внешних статистических* переменных, но это слегка запутывает, т.к. они имеют внутреннее связывание. К сожалению, новый компактный термин найти не удалось, поэтому нам остается вариант *статическая переменная с внутренним связыванием*. Обычная внешняя переменная может использоваться функциями в любом файле, который является частью программы, но статическая переменная с внутренним связыванием может применяться только функциями в том же самом файле. Внутри функции можно повторно объявить любую переменную с областью видимости в пределах файла, используя спецификатор класса хранения extern. Такое объявление не изменяет тип связывания. Рассмотрим следующий код:

```
int traveler = 1;          // внешнее связывание
static int stayhome = 1;   // внутреннее связывание
int main()
{
   extern int traveler;    // использование глобальной переменной traveler
   extern int stayhome;    // использование глобальной переменной stayhome
   ...
```

Переменные traveler и stayhome являются глобальными в этой конкретной единице трансляции, но только traveler можно применять в других единицах трансляции. Два объявления, использующих extern, документируют тот факт, что в main() применяются две глобальных переменных, но stayhome продолжает иметь внутреннее связывание.

Множество файлов

Отличие между внутренним и внешним связыванием важно только в ситуации, когда программа строится из нескольких единиц трансляции, поэтому давайте кратко рассмотрим данную тему.

Сложные программы на C часто состоят из нескольких отдельных файлов исходного кода. Иногда в этих файлах возникает необходимость совместного использования какой-то внешней переменной. Чтобы сделать это в C, необходимо предусмотреть определяющее объявление в одном файле и ссылочные объявления в остальных файлах. Это значит, что во всех объявлениях кроме одного (определяющего объявления) должно присутствовать ключевое слово extern, а для инициализации переменной должно применяться только определяющее объявление.

Обратите внимание, что внешняя переменная, определенная в одном файле, не будет доступна во втором файле до тех пор, пока ее также там не объявить (с использованием extern). Само по себе внешнее объявление лишь делает переменную потенциально доступной другим файлам.

Однако исторически сложилось так, что многие компиляторы в этом отношении следуют другим правилам. Например, многие системы Unix позволяют объявлять переменную в нескольких файлах без указания ключевого слова extern при условии, что только одно объявление включает инициализацию. Объявление с инициализацией считается определением.

Спецификаторы классов хранения

Вы могли уже заметить, что смысл ключевых слов static и extern зависит от контекста. В языке С имеется шесть ключевых слов, которые сгруппированы вместе как спецификаторы классов хранения: auto, register, static, extern, _Thread_local и typedef. Ключевое слово typedef ничего не говорит о хранении в памяти, но оно присутствует здесь по синтаксическим причинам. Например, в большинстве случаев вы можете использовать в объявлении не более одного спецификатора класса хранения, а это означает, что вы не можете применять один из спецификаторов класса хранения в качестве части typedef. Исключением является спецификатор _Thread_local, который можно использовать вместе со спецификаторами static и extern.

Спецификатор auto указывает переменную с автоматической продолжительностью хранения. Он может применяться только в объявлениях переменных с областью видимости в пределах блока, которые уже имеют автоматическую продолжительность хранения, так что главным его предназначением является документирование.

Спецификатор register также может использоваться только с переменными, имеющими область видимости в пределах блока. Он помещает переменную в регистровый класс хранения, что равносильно запросу на минимизацию времени доступа к ней. Он также предотвращает взятие адреса этой переменной.

Спецификатор static создает объект со статической продолжительностью хранения, который появляется после загрузки программы в память и исчезает при завершении программы. Если static применяется в объявлении с областью видимости в пределах файла, то область видимости ограничивается одним этим файлом. Если static используется в объявлении с областью видимости в пределах блока, то область видимости ограничивается этим блоком. Таким образом, объект существует и сохраняет свое значение на протяжении выполнения программы, но может быть доступен посредством идентификатора, только когда выполняется код внутри его блока. Статическая переменная с областью видимости в пределах блока не имеет связывания. Статическая переменная с областью видимости в пределах файла имеет внутреннее связывание.

Спецификатор extern указывает, что вы объявляете переменную, которая была определена в каком-то другом месте. Если объявление, содержащее extern, имеет область видимости в пределах файла, то переменная, на которую производится ссылка, должна иметь внешнее связывание. Если объявление с extern имеет область видимости в пределах блока, то ссылаемая переменная может иметь либо внешнее, либо внутреннее связывание, что зависит от определяющего объявления этой переменной.

Сводка: классы хранения

Автоматические переменные имеют область видимости в пределах блока, не имеют связывания и характеризуются автоматической продолжительностью хранения. Эти переменные локальны и закрыты для блока (обычно функции), в котором они определены. Регистровые переменные обладают такими же свойствами, как автоматические переменные, но для их хранения компилятор может применять более быструю память или регистры. Адрес регистровой переменной получать нельзя.

Переменные со статической продолжительностью хранения могут иметь внешнее связывание, внутреннее связывание или вообще не иметь связывания. Когда переменная объявляется внешней по отношению к любой функции в файле, то она представляет собой внешнюю переменную и имеет область видимости в пределах файла, внешнее связывание и статическую продолжительность хранения.

Если вы добавите к такому объявлению ключевое слово `static`, то получите переменную со статической продолжительностью хранения, областью видимости в пределах файла и внутренним связыванием. Если вы объявляете переменную внутри функции и указываете ключевое слово `static`, то данная переменная получает статическую продолжительность хранения, область видимости в пределах блока и отсутствие связывания.

Память для переменной с автоматической продолжительностью хранения выделяется, когда поток управления входит в блок, содержащий объявление переменной, и освобождается после покидания этого блока потоком управления. Неинициализированная переменная такого рода имеет случайное значение. Память для переменной со статической продолжительностью хранения выделяется на этапе компиляции и сохраняется на все время выполнения программы. Неинициализированная переменная такого рода получает значение 0.

Переменная с областью видимости в пределах блока является локальной по отношению к блоку, содержащему объявление. Переменная с областью видимости в пределах файла известна всем функциям в файле (или единице трансляции), которые находятся после ее объявления. Если переменная с областью видимости в пределах файла имеет внешнее связывание, то она может использоваться другими единицами трансляции в программе. Если переменная с областью видимости в пределах файла имеет внутреннее связывание, то она может применяться только внутри файла, в котором она объявлена.

Ниже показана короткая программа, в которой используются все классы хранения. Код разнесен на два файла (листинг 12.5 и листинг 12.6), так что вы должны провести многофайловую компиляцию. (За деталями обращайтесь в главу 9 или в руководство по компилятору.) Главная цель программы заключается в демонстрации всех классов хранения, а не в том, чтобы предложить проектную модель; в качественном проекте нет нужды в переменных с областью видимости в пределах файла.

Листинг 12.5. Программа `parta.c`

```
// parta.c -- разнообразные классы хранения
// компилировать вместе с partb.c
#include <stdio.h>
void report_count();
void accumulate(int k);
int count = 0;       // область видимости в пределах файла, внешнее связывание

int main(void)
{
    int value;       // автоматическая переменная
    register int i;  // регистровая переменная

    printf("Введите положительное целое число (0 для завершения): ");
    while (scanf("%d", &value) == 1 && value > 0)
    {
        ++count;     // использование переменной с областью видимости в пределах файла
        for (i = value; i >= 0; i--)
            accumulate(i);
        printf("Введите положительное целое число (0 для завершения): ");
    }
    report_count();

    return 0;
}

void report_count()
{
    printf("Цикл выполнен %d раз(а)\n", count);
}
```

Листинг 12.6. Программа partb.c

```
// partb.c -- остальная часть программы
// компилировать вместе с parta.c
#include <stdio.h>
extern int count;               // ссылочное объявление, внешнее связывание
static int total = 0;           // статическое определение, внутреннее связывание
void accumulate(int k);         // прототип

void accumulate(int k)          // k имеет область видимости в пределах блока,
                                // связывание отсутствует
{
   static int subtotal = 0;     // статическая переменная, связывание отсутствует
   if (k <= 0)
   {
      printf("итерация цикла: %d\n", count);
      printf("subtotal: %d; total: %d\n", subtotal, total);
      subtotal = 0;
   }
   else
   {
      subtotal += k;
      total += k;
   }
}
```

В этой программе статическая переменная с областью видимости в пределах блока, имеющая имя subtotal, накапливает промежуточную сумму значений, передаваемых функции accumulate(), а переменная total с областью видимости в пределах файла и внутренним связыванием накапливает общую сумму. Функция accumulate() выводит значения total и subtotal каждый раз, когда ей передается неположительное значение; в таких ситуациях она также сбрасывает subtotal в 0. Прототип accumulate() в программе parta.c обязателен, т.к. файл содержит вызов функции accumulate(). В файле partb.c прототип не обязателен, поскольку функция в нем определена, но не вызывается. В этой функции также применяется внешняя переменная count для отслеживания количества итераций цикла while, выполненных в main(). (Кстати, это хороший пример того, как не следует использовать внешнюю переменную, потому что она нежелательным образом переплетает код в parta.c с кодом в partb.c.) В файле parta.c функции main() и report_count() совместно осуществляют доступ к count.

Вот результаты пробного запуска:

```
Введите положительное целое число (0 для завершения): 5
итерация цикла: 1
subtotal: 15; total: 15
Введите положительное целое число (0 для завершения): 10
итерация цикла: 2
subtotal: 55; total: 70
Введите положительное целое число (0 для завершения): 2
итерация цикла: 3
subtotal: 3; total: 73
Введите положительное целое число (0 для завершения): 0
Цикл выполнен 3 раз(а)
```

Классы хранения и функции

Функции также имеют классы хранения. Функция может быть либо внешней (по умолчанию), либо статической. (В стандарте C99 добавлена третья возможность — встраиваемая функция, которая обсуждается в главе 16.) Доступ к внешней функции могут получать функции в других файлах, но статическая функция может применяться только внутри файла, где она определена. Рассмотрим, например, файл со следующими прототипами функций:

```
double gamma(double);       /* по умолчанию внешняя */
static double beta(int, int);
extern double delta(double, int);
```

Функции `gamma()` и `delta()` могут использоваться функциями в других файлах, которые являются частью программы, но `beta()` — нет. Из-за такого ограничения функции `beta()` одним файлом в остальных файлах можно применять другие функции с этим же именем. Причина использования класса хранения `static` связана с созданием функций, закрытых в отношении конкретного модуля, благодаря чему устраняется возможность конфликта имен.

Обычная практика предусматривает применение ключевого слова `extern` при объявлении функции, определенной в другом файле. Главным образом это касается ясности, т.к. объявление функции предполагается как `extern`, если только не указано ключевое слово `static`.

Выбор класса хранения

Ответом на вопрос о том, какой выбрать класс хранения, чаще всего будет — автоматический. В конце концов, по какой еще причине автоматический класс хранения был выбран по умолчанию? Да, мы знаем, что на первый взгляд внешний класс хранения выглядит более привлекательным. Стоит лишь сделать все свои переменные внешними, и не придется беспокоиться об использовании аргументов и указателей при взаимодействии между функциями.

Однако здесь подстерегает коварная ловушка. Вам придется переживать о том, что функция `A()` незаметно изменит значения переменных, применяемых в функции `B()`, хотя ваши намерения были совершенно другими. Несомненное свидетельство несметного количества лет, на протяжении которых формировался коллективный опыт программистов, говорит о том, что одна эта скрытая опасность далеко превосходит сомнительную привлекательность неразборчивого использования переменных с внешним классом хранения.

Распространенным исключением из правила являются данные `const`. Поскольку они не могут быть изменены, нет нужды переживать по поводу их непреднамеренной модификации:

```
const int DAYS = 7;
const char * MSGS[3] = {"Да", "Нет", "Возможно"};
```

Одним из золотых правил защитного программирования является принцип "необходимого знания". Держите всю внутреннюю работу каждой функции максимально закрытой в рамках этой функции, совместно используя только те переменные, которые должны совместно использоваться. Другие классы хранения удобны, и они доступны. Однако прежде чем выбирать какой-то из них, подумайте, есть ли в этом необходимость.

Функция генерации случайных чисел и статическая переменная

Теперь, когда вы получили необходимый минимум знаний о классах хранения, давайте рассмотрим пару программ, в которых они применяются. Первым делом мы взглянем на функцию, которая использует статическую переменную с внутренним связыванием: функцию генерации случайных чисел. Для генерации случайных чисел библиотека ANSI C предлагает функцию rand(). Существуют разнообразные алгоритмы генерации случайных чисел, и ANSI C позволяет реализациям выбирать наилучший алгоритм для конкретной машины. Однако ANSI C также предлагает стандартный переносимый алгоритм, который выдает те же самые случайные числа в разных системах. В действительности функция rand() является "генератором псевдослучайных чисел", т.е. фактическая последовательность чисел предсказуема, но числа достаточно равномерно распределены по диапазону возможных значений.

Вместо применения встроенной функции rand() компилятора, мы будем использовать переносимую версию ANSI, чтобы вы могли видеть, что происходят внутри. Схема начинается с числа, которое называется "начальным". Функция применяет начальное число для получения нового числа, которое становится новым начальным числом. Затем новое начальное число может использоваться для получения следующего нового начального числа и т.д. Чтобы эта схема работала, функция генерации случайных чисел должна запоминать начальное число, которое применялось при ее последнем вызове. Здесь и возникает потребность в статической переменной. В листинге 12.7 представлена версия 0 (вскоре появится и версия 1).

Листинг 12.7. Функция rand0.c

```
/* rand0.c -- генерация случайных чисел           */
/*           используется переносимый алгоритм ANSI C */
static unsigned long int next = 1;  /* начальное число */

int rand0(void)
{
   /* магическая формула генерации псевдослучайных чисел */
   next = next * 1103515245 + 12345;
   return (unsigned int) (next/65536) % 32768;
}
```

В листинге 12.7 статическая переменная next начинает со значения 1 и изменяется магической формулой при каждом вызове функции. Результатом будет возвращаемое значение, находящееся где-то в диапазоне от 0 до 32767. Обратите внимание, что next является статической переменной с внутренним связыванием, а не просто статической переменной без связывания. Дело в том, что позже пример будет расширен, чтобы переменная next совместно использовалась двумя функциями в том же самом файле.

Давайте протестируем функцию rand0() с помощью простого драйвера, показанного в листинге 12.8.

Листинг 12.8. Драйвер r_drive0.c

```
/* r_drive0.c -- тестирование функции rand0() */
/* компилировать вместе с rand0.c           */
#include <stdio.h>
extern int rand0(void);
```

```
int main(void)
{
   int count;

   for (count = 0; count < 5; count++)
      printf("%d\n", rand0());

   return 0;
}
```

Здесь вы имеете еще один шанс попрактиковаться с применением нескольких файлов. Поместите код из листинга 12.7 в один файл, а код из листинга 12.8 — в другой. Ключевое слово extern напоминает, что функция rand0() определена в отдельном файле, хотя оно необязательно.

Вывод имеет следующий вид:

```
16838
5758
10113
17515
31051
```

Выходные данные производят впечатление случайных, но давайте запустим программу снова. И вот какие результаты получены на этот раз:

```
16838
5758
10113
17515
31051
```

Числа выглядят знакомыми; в этом и заключается аспект "псевдо". Каждый раз, когда главная программа запускается, старт происходит с одного и того же начального числа 1. Проблему можно обойти путем ввода второй функции по имени srand1(), которая позволит переустанавливать начальное число. Трюк заключается в том, чтобы сделать next статической переменной с внутренним связыванием, которая известна только функциям rand1() и srand1(). (Эквивалент srand1() в библиотеке C называется srand().) Добавьте функцию srand1() в файл, содержащий rand1().

В листинге 12.9 представлена модифицированная версия.

Листинг 12.9. Программа s_and_r.c

```
/* s_and_r.c -- файл для функций rand1() и srand1()      */
/*              используется переносимый алгоритм ANSI C */
static unsigned long int next = 1;  /* начальное число  */

int rand1(void)
{
   /* магическая формула для генерации псевдослучайных чисел */
   next = next * 1103515245 + 12345;
   return (unsigned int) (next/65536) % 32768;
}

void srand1(unsigned int seed)
{
   next = seed;
}
```

Обратите внимание, что next — это статическая переменная с областью видимости в пределах файла и внутренним связыванием. Это означает, что она может использоваться как randl (), так и srandl (), но не функциями в других файлах. Для тестирования этих функций применяйте драйвер из листинга 12.10.

Листинг 12.10. Драйвер r_drivel.c

```
/* r_drivel.c -- тестирование функций randl() и srandl() */
/* компилировать вместе с s_and_r.c                      */
#include <stdio.h>
extern void srandl(unsigned int x);
extern int randl(void);

int main(void)
{
   int count;
   unsigned seed;

   printf("Введите желаемое начальное число.\n");
   while (scanf("%u", &seed) == 1)
   {
      srandl(seed);    /* переустановка начального числа */
      for (count = 0; count < 5; count++)
         printf("%d\n", randl());
      printf("Введите следующее начальное число (q для завершения):\n");
   }
   printf("Программа завершена.\n");

   return 0;
}
```

Снова скомпилируйте два файла и запустите программу:

```
Введите желаемое начальное число.
1
16838
5758
10113
17515
31051
Введите следующее начальное число (q для завершения):
513
20067
23475
8955
20841
15324
Введите следующее начальное число (q для завершения):
q
Программа завершена.
```

Использование значения 1 для seed выдает те же псевдослучайные числа, что и ранее, но значение 3 для seed обеспечивает выдачу новых результатов.

НА ЗАМЕТКУ! Автоматическая переустановка начального значения

Если ваша реализация C предоставляет доступ к какой-то меняющейся величине, такой как показание системных часов, ее можно применять (возможно, с усечением) для инициализации начального числа.

Например, в ANSI C имеется функция time(), которая возвращает системное время. Единицы измерения времени зависят от системы, однако здесь важно то, что возвращаемое значение имеет арифметический тип и меняется с течением времени. Точный тип может отличаться от системы к системе, и он получил метку time_t, но можно воспользоваться приведением. Вот как выглядит базовый подход:

```
#include <time.h>                    /* прототип ANSI для time() */
srandl((unsigned int) time(0));      /* инициализация начального числа */
```

В общем time() принимает аргумент, который является адресом объекта типа time_t. В данном случае значение времени также сохраняется по этому адресу. В качестве аргумента можно передать нулевой указатель (0), тогда значение будет передаваться только с помощью механизма возврата из функции.

Вы можете применять тот же самый метод с функциями srand() и rand() из ANSI C. Если вы планируете использовать эти функции, включите заголовочный файл stdlib.h. Фактически теперь, когда вы увидели, каким образом в srandl() и randl() применяется статическая переменная с внутренним связыванием, вы можете также воспользоваться версиями, которые поставляет компилятор. Мы сделаем это в следующем примере.

Игра в кости

Мы собираемся эмулировать весьма популярное действие со случайным характером – игру в кости. В наиболее распространенной форме этой игры участвуют две шестигранных кости, но существуют и другие разновидности. Во многих азартных играх применяются все пять геометрически возможных костей – с 4, 6, 8, 12 и 20 гранями. Талантливые древние греки доказали, что лишь у пяти правильных тел все грани имеют одинаковую форму и размер, и эти тела являются основой для всего разнообразия костей. Можно было бы сделать кости с другим числом граней, но не все грани имели бы одинаковый размер, что не способствовало бы уравниванию шансов их выпадения.

Компьютерные вычисления не ограничиваются этими геометрическими соображениями, так что мы можем придумать электронную игральную кость, которая имеет любое количество граней. Давайте начнем с варианта с шестью гранями, после чего займемся обобщением. Нам необходимо случайное значение от 1 до 6. Тем не менее, функция rand() генерирует целое число из диапазона от 0 до RAND_MAX; значение RAND_MAX определено в stdlib.h. Обычно это INT_MAX. Следовательно, понадобятся провести ряд настроек. Ниже представлен один из подходов.

1. Получить случайное число по модулю 6. Это даст целое число из диапазона от 0 до 5.
2. Добавить к нему 1. Новое число находится в диапазоне от 1 до 6.
3. Для обобщения этого алгоритма просто замените 6 в первом шаге количеством граней.

Описанные идеи реализованы в следующем коде:

```
#include <stdlib.h>   /* для rand() */
int rollem(int sides)
{
   int roll;
   roll = rand() % sides + 1;
   return roll;
}
```

Давайте немного расширим возможности и сделаем так, чтобы функция позволяла бросать произвольное количество костей и возвращала общее число очков. Это реализовано в листинге 12.11.

Листинг 12.11. Файл `diceroll.c`

```
/* diceroll.c -- эмуляция игры в кости */
/* компилировать вместе с mandydice.c  */
#include "diceroll.h"
#include <stdio.h>
#include <stdlib.h>          /* для библиотечной функции rand() */

int roll_count  = 0;         /* внешнее связывание              */

static int rollem(int sides) /* закрытая для этого файла        */
{
   int roll;

   roll = rand() % sides + 1;
   ++roll_count;             /* подсчет вызовов функции         */

   return roll;
}

int roll_n_dice(int dice, int sides)
{
   int d;
   int total = 0;
   if (sides < 2)
   {
      printf("Требуются, по меньшей мере, 2 грани.\n");
      return -2;
   }
   if (dice < 1)
   {
      printf("Требуется, по меньшей мере, 1 кость.\n");
      return -1;
   }

   for (d = 0; d < dice; d++)
      total += rollem(sides);

   return total;
}
```

В этом файле предпринято несколько действий. Во-первых, `rollem()` сделана функцией, закрытой для файла. Она выступает в качестве вспомогательной функции для `roll_n_dice()`. Во-вторых, для демонстрации работы внешнего связывания в файле объявлена внешняя переменная по имени `roll_count`, которая отслуживает количество вызовов функции `rollem()`. Пример несколько надуман, но он показывает, как работают внешние переменные.

В-третьих, файл содержит следующий оператор:

```
#include "diceroll.h"
```

В случае использования стандартных библиотечных функций, таких как `rand()`, вы включаете стандартный заголовочный файл (`stdlib.h` для `rand()`) вместо объявления функции. Причина в том, что такой заголовочный файл уже содержит корректное объявление. Мы эмулируем этот подход, предоставляя заголовочный файл `diceroll.h` для применения функции `roll_n_dice()`. Заключение имени файла в двойные кавычки, а не в угловые скобки, указывает компилятору на необходимость

поиска этого файла локально, а не в стандартных местоположениях, которые используются для хранения стандартных заголовочных файлов. Смысл выражения "поиска локально" зависит от реализации. Распространенные интерпретации предполагают помещение заголовочного файла в тот же каталог или папку, где находится исходный код или файл проекта (если ваш компилятор имеет дело с ним). Содержимое заголовочного файла приведено в листинге 12.12.

Листинг 12.12. Файл diceroll.h

```
// diceroll.h
extern int roll_count;

int roll_n_dice(int dice, int sides);
```

Данный заголовочный файл содержит прототипы функций и объявление extern. Поскольку diceroll.c включает этот заголовочный файл, diceroll.c в действительности содержит два объявления переменной roll_count:

```
extern int roll_count;    // из заголовочного файла
int roll_count = 0;       // из файла исходного кода
```

Можно иметь только одно определяющее объявление переменной. Однако объявление с ключевым словом extern является ссылочным, и таких объявлений может быть столько, сколько пожелаете.

Программа, в которой применяется функция roll_n_dice(), должна включать указанный заголовочный файл. Это не только предоставляет прототип функции roll_n_dice(), но также делает доступной переменную roll_count в программе. Все сказанное иллюстрируется в листинге 12.13.

Листинг 12.13. Файл manydice.c

```
/* manydice.c -- бросание множества костей                    */
/* компилировать вместе с diceroll.c                          */
#include <stdio.h>
#include <stdlib.h>          /* для библиотечной функции srand() */
#include <time.h>            /* для функции time()              */
#include "diceroll.h"        /* для функции roll_n_dice()       */
                             /* и для переменной roll_count     */
int main(void)
{
   int dice,roll;
   int sides;

   srand((unsigned int) time(0)); /* рандомизация начального числа */
   printf("Введите количество граней кости или 0 для завершения программы.\n");
   while (scanf("%d", &sides) == 1 && sides > 0 )
   {
      printf("Сколько костей?\n");
      if ((status =scanf("%d", &dice)) != 1)
      {
         if (status == EOF)
            break;           /* выход из цикла           */
         else
         {
            printf("Вы должны ввести целое число.");
            printf(" Давайте начнем с начала.\n");
            while (getchar() != '\n')
               continue;     /* отбросить некорректный ввод */
```

```
            printf("Сколько граней? Для прекращения введите 0.\n");
            continue;        /* новая итерация цикла      */
         }
      }
      roll = roll_n_dice(dice, sides);
      printf("Вы бросали %d раз(а), используя %d кости с %d гранями.\n",
             roll, dice, sides);
      printf("Сколько граней? Для прекращения введите 0.\n");
   }
   printf("Функция rollem() была вызвана %d раз(а).\n",
          roll_count);           /* используется переменная extern */
   printf("Пусть удача не покидает вас!\n");
   return 0;
}
```

Скомпилируйте файл с кодом из листинга 12.13 вместе с файлом, содержащим код из листинга 12.11. Для простоты поместите файлы с исходным кодом из листингов 12.11, 12.12 и 12.13 в одну и ту же папку или каталог. Запустите результирующую программу. Вывод должен иметь примерно такой вид:

```
Введите количество граней кости или 0 для завершения программы.
6
Сколько костей?
2
Вы бросали 12 раз(а), используя 2 кости с 6 гранями.
Сколько граней? Для прекращения введите 0.
6
Сколько костей?
2
Вы бросали 4 раз(а), используя 2 кости с 6 гранями.
Сколько граней? Для прекращения введите 0.
6
Сколько костей?
2
Вы бросали 5 раз(а), используя 2 кости с 6 гранями.
Сколько граней? Для прекращения введите 0.
0
Функция rollem() была вызвана 6 раз(а).
Пусть удача не покидает вас!
```

Из-за того, что в программе используется функция `srand()` для рандомизации начального случайного числа, скорее всего, вы не получите один и тот же вывод при том же самом входном значении. Обратите внимание, что функция `main()` в `manydice.c` имеет доступ к переменной `roll_count`, определенной в `diceroll.c`.

Внешний цикл `while` может прекратиться по трем причинам: значение `sides` оказывается меньше 1, введенное значение не соответствует по типу (`scanf()` возвращает 0) или встретился конец файла (`scanf()` возвращает `EOF`). При чтении количества костей появление конца файла обрабатывается иначе, чем ситуация с несоответствием типа; в первом случае происходит выход из цикла `while`, а во втором инициируется новая итерация цикла.

Функцию `roll_n_dice()` можно применять многими способами. В случае значения `sides`, равного 2, программа эмулирует процесс бросания монеты, при котором выпадение орла обозначается 2, а решки — 1 (или, если хотите, то наоборот). Вы можете легко модифицировать программу так, чтобы она показывала отдельные результаты и сумму, или построить эмулятор игры в кости. Если требуется большое коли-

чество бросаний, как в некоторых ролевых играх, несложно изменить программу для получения примерного такого вывода:

```
Введите количество бросаний или q для завершения.
18
Сколько граней и сколько костей?
6 3
Имеем 18 бросаний 3 костей с 6 гранями.
  12  10   6   9   8  14   8  15   9  14  12  17  11   7  10
  13   8  14
Введите количество бросаний или q для завершения.
q
```

Функции randl() и rand() (но не rollem()) могут также использоваться для создания программы, в которой компьютер выбирает число, а вы его отгадываете. Попробуйте написать ее самостоятельно.

Выделенная память: malloc() и free()

Рассмотренные классы хранения имеют одну общую особенность. После выбора класса хранения автоматически появятся решения относительно области видимости и продолжительности хранения. Варианты подчиняются предварительно укомплектованным правилам управления памятью. Однако на выбор доступен еще один вариант, который обеспечивает более высокую гибкость. Он предусматривает применение библиотечных функций выделения и управления памятью.

Для начала давайте вспомним ряд фактов, касающихся выделения памяти. Все программы должны резервировать пространство памяти, достаточное для хранения данных, с которыми они работают. Некоторые операции по выделению памяти происходят автоматически. Например, в результате объявлений

```
float x;
char place[] = "Поющие в терновнике";
```

резервируется пространство памяти, достаточное для хранения переменной float и строки. Можно также явно запросить определенный объем памяти:

```
int plates[100];
```

Это объявление резервирует 100 ячеек памяти, в каждой из которых можно хранить значение int. Во всех показанных случаях объявление также предоставляет идентификатор выделенной памяти, так что для обращения к данным можно использовать x или place. Как вы помните, память под статические данные выделяется во время загрузки программы в память, а память под автоматические данные выделяется, когда поток управления программы входит в блок, и освобождается, когда поток управления покидает блок.

Язык С выходит за эти рамки. Во время выполнения программы можно выделять дополнительную память. Основным инструментом является функция malloc(), которая принимает один аргумент: нужное количество байтов памяти. Затем malloc() ищет подходящий блок свободной памяти. Память будет анонимной, т.е. функция malloc() выделяет блок памяти, но не назначает ему имя. Тем не менее, она возвращает адрес первого байта в этом блоке. Следовательно, вы можете присвоить этот адрес переменной типа указателя и применять такой указатель для доступа в память. Поскольку байт представлен посредством char, функция malloc() традиционно была объявлена с типом указателя на char. Однако в стандарте ANSI С используется новый тип: указатель на void. Этот тип задумывался как обобщенный указатель.

Функция malloc() может применяться для возвращения указателей на массивы, структуры и т.п., поэтому обычно возвращаемое значение приводится к подходящему типу. В условиях стандарта ANSI С для ясности вы по-прежнему должны осуществлять приведение, но присваивание значения указателя на void указателю другого типа не считается конфликтом типов. Если функции malloc() не удается найти запрошенное пространство, она возвращает нулевой указатель.

Давайте воспользуемся malloc() для решения задачи создания массива. С помощью malloc() можно запросить блок памяти во время выполнения программы. Кроме того, понадобится указатель, чтобы отслеживать, где в памяти находится выделенный блок. Например, взгляните на следующий код:

```
double * ptd;
ptd = (double *) malloc(30 * sizeof(double));
```

Этот код запрашивает пространство под 30 значений типа double и устанавливает ptd для указания на соответствующую ячейку памяти. Обратите внимание, что ptd объявлен как указатель на одиночное значение double, а не на блок из 30 значений double. Вспомните, что имя массива представляет собой адрес его первого элемента. Следовательно, если вы установили ptd так, чтобы он указывал на первый элемент блока, вы можете использовать его подобно имени массива. То есть вы можете применять выражение ptd[0] для доступа к первому элементу блока, ptd[1] — для доступа ко второму элементу и т.д. Как было показано ранее, форму записи с указателями можно использовать для имен массивов, а форму записи с массивами можно применять для указателей.

Теперь у вас есть три способа создания массива.

- Объявить массив, используя константные выражения для размерностей, и применять для доступа к элементам имя массива. Такой массив может быть создан с использованием либо статической, либо автоматической памяти.
- Объявить массив переменной длины, применяя переменные выражения для размерностей, и использовать для доступа к элементам имя массива. (Вспомните, что эта возможность предусмотрена стандартом С99.) Такой вариант доступен только для автоматической памяти.
- Объявить указатель, вызвать malloc(), присвоить возвращаемое значение указателю и применять для доступа к элементам указатель. Этот указатель может быть либо статическим, либо автоматическим.

Второй и третий методы можно использовать для выполнения того, что не получится сделать с обычным объявленным массивом — создать *динамический массив*, память под который выделяется во время выполнения программы и тогда же есть возможность выбрать его размер. Предположим, например, что n — целочисленная переменная. До выхода стандарта С99 нельзя было поступать так:

```
double item[n];    /* до C99: не разрешено, если n является переменной */
```

Однако можно было записывать следующим образом даже в случае компилятора, выпущенного до выхода С99:

```
ptd = (double *) malloc(n * sizeof(double));  /* нормально */
```

Этот прием работает и, как вы вскоре убедитесь, он обладает несколько большей гибкостью, чем массив переменной длины.

Обычно вы должны компенсировать каждый случай вызова malloc() вызовом free(). Функция free() принимает в качестве аргумента адрес, возвращенный ранее

функцией malloc(), и освобождает память, которая была выделена. Таким образом, продолжительность существования выделенной памяти рассчитывается с момента, когда была вызвана функция malloc() для выделения памяти, и до момента, когда вызывается функция free() с целью освобождения памяти для ее повторного использования. Функции malloc() и free() можно рассматривать как инструменты для управления пулом памяти. Каждый вызов malloc() выделяет память для применения программой, а каждый вызов free() восстанавливает память в пуле, так что она может повторно использоваться. Аргументом free() должен быть указатель на блок памяти, выделенный malloc(); функцию free() нельзя применять для освобождения памяти, выделенной другими средствами, такими как объявление массива. Функции malloc() и free() имеют прототипы в заголовочном файле stdlib.h.

За счет использования malloc() программа может решать, массив какого размера требуется, и создавать его во время выполнения. Эта возможность демонстрируется в листинге 12.14. В нем указателю ptd присваивается адрес блока памяти, после чего ptd применяется, как если бы это было имя массива. Если выделить нужную память не удалось, для прекращения работы программы вызывается функция exit(), прототип которой содержится в stdlib.h. Значение EXIT_FAILURE определено в этом же заголовочном файле. Стандарт предоставляет два возвращаемых значения, которые гарантированно распознают все операционные системы: EXIT_SUCCESS (эквивалентно значению 0) для указания на нормальное завершение программы и EXIT_FAILURE для указания на аварийное завершение. Некоторые операционные системы, включая Unix, Linux и Windows, могут принимать дополнительные целочисленные значения, обозначающие конкретные формы отказа.

Листинг 12.14. Программа dyn_arr.c

```
/* dyn_arr.c -- динамически выделяемый массив */
#include <stdio.h>
#include <stdlib.h> /* для malloc(), free() */

int main(void)
{
   double * ptd;
   int max = 0;
   int number;
   int i = 0;

   puts("Введите максимальное количество элементов типа double.");
   if (scanf("%d", &max) != 1)
   {
      puts("Количество введено некорректно -- программа завершена.");
      exit(EXIT_FAILURE);
   }
   ptd = (double *) malloc(max * sizeof (double));
   if (ptd == NULL)
   {
      puts("Не удалось выделить память. Программа завершена.");
      exit(EXIT_FAILURE);
   }
   /* ptd теперь указывает на массив из max элементов */
   puts("Введите значения (q для выхода):");
   while (i < max && scanf("%lf", &ptd[i]) == 1)
      ++i;
   printf("Введено %d элементов:\n", number = i);
   for (i = 0; i < number; i++)
   {
```

```
        printf("%7.2f ", ptd[i]);
        if (i % 7 == 6)
            putchar('\n');
    }
    if (i % 7 != 0)
        putchar('\n');
    puts("Программа завершена.");
    free(ptd);

    return 0;
}
```

Ниже показаны результаты пробного запуска. Мы ввели шесть чисел, но программа обработала только пять из них, поскольку размер массива был ограничен до 5.

```
Введите максимальное количество элементов типа double.
5
Введите значения (q для выхода):
20 30 35 25 40 80
Введено 5 элементов:
  20.00   30.00   35.00   25.00   40.00
Программа завершена.
```

Давайте рассмотрим код. Программа получает нужный размер массива с помощью следующих строк:

```
if (scanf("%d", &max) != 1)
{
    puts("Количество введено некорректно -- программа завершена.");
    exit(EXIT_FAILURE);
}
```

Показанная ниже строка кода выделяет в памяти пространство, достаточное для хранения запрошенного количества элементов, и затем присваивает адрес этого блока указателю `ptd`:

```
ptd = (double *) malloc(max * sizeof (double));
```

Приведение к `(double *)` не обязательно в C, но требуется в C++, поэтому использование приведения типа упрощает перенос программы из C в C++.

Вполне вероятно, что функция `malloc()` не сможет предоставить желаемый объем памяти. В этом случае она возвращает нулевой указатель, и выполнение программы прекращается:

```
if (ptd == NULL)
{
    puts("Не удалось выделить память. Программа завершена.");
    exit(EXIT_FAILURE);
}
```

Если программа преодолеет это препятствие, она может трактовать `ptd`, как если бы оно было именем массива из `max` элементов, что и делается.

Обратите внимание на вызов функции `free()` ближе к концу программы. Она освобождает память, выделенную `malloc()`. Функция `free()` освобождает только блок памяти, на который указывает ее аргумент. Некоторые операционные системы будут освобождать выделенную память автоматически при завершении программы, но другие могут этого не делать. Таким образом, применяйте `free()` и не полагайтесь на то, что операционная система выполнит очистку вместо вас.

Какую пользу мы извлекли из того, что воспользовались динамическим массивом? В этом случае мы увеличили гибкость программы. Предположим, вы знаете, что большую часть времени программе будет требоваться не более 100 элементов, но иногда она будет нуждаться в 10 000 элементов. Если вы объявляете массив, то должны учитывать худший случай и объявить его с 10 000 элементов. Большую часть времени программа будет расходовать память понапрасну. К тому же, если наступит момент, когда понадобится иметь 10 001 элемент, программа потерпит отказ. Применение динамического массива позволяет программе подстраиваться под существующие обстоятельства.

Важность функции free()

Объем статической памяти фиксируется во время компиляции; он не изменяется на протяжении выполнения программы. Объем памяти, используемой для автоматических переменных, растет и убывает автоматически по мере выполнения программы. Однако если вы забудете вызывать функцию free(), то объем выделенной памяти будет только расти. Например, предположим, что имеется функция, которая создает временную копию массива, как схематично продемонстрировано в следующем коде:

```
...
int main()
{
   double glad[2000];
   int i;
   ...
   for (i = 0; i < 1000; i++)
      gobble(glad, 2000);
   ...
}
void gobble(double ar[], int n)
{
   double * temp = (double *) malloc( n * sizeof(double));
   ...  /* free(temp); // забыли воспользоваться free() */
}
```

Когда функция gobble() вызывается в первый раз, она создает указатель temp и применяет malloc() для выделения 16 000 байтов памяти (мы исходим из предположения, что тип double занимает 8 байтов). Представим, что мы не вызвали free(), как показано в коде. Когда gobble() завершится, указатель temp, будучи автоматической переменной, исчезает. Но 16 000 байтов памяти, на которые он указывал, по-прежнему существуют. Доступ к этой памяти невозможен, т.к. мы больше не располагаем ее адресом. Она не может использоваться повторно, потому что не была вызвана функция free().

Когда функция gobble() вызывается во второй раз, она снова создает temp и вызывает malloc() для выделения 16 000 байтов памяти. Первый блок из 16 000 байтов недоступен, поэтому функция malloc() должна найти второй блок размером 16 000 байтов. Когда функция завершится, этот блок памяти также становится недоступным и не может использоваться повторно.

Цикл повторяется 1000 раз, и ко времени его завершения из пула памяти будет изъято 16 000 000 байтов. На самом деле программе может просто не хватить памяти, чтобы зайти настолько далеко. Проблема подобного рода называется *утечкой памяти* и может быть предотвращена за счет наличия вызова free() в конце функции.

Функция calloc()

Еще один способ выделения памяти предусматривает применение функции calloc(). Ниже показан типичный случай ее использования:

```
long * newmem;
newmem = (long *)calloc(100, sizeof (long));
```

Подобно malloc(), функция calloc() возвращает указатель на char в своей версии до выхода стандарта ANSI и указатель на void в условиях действия стандарта ANSI. Если необходим другой тип, вы должны применять приведение. Эта новая функция принимает два аргумента, которые оба должны быть целыми числами без знака (типа size_t в ANSI). Первый аргумент задает желаемое количество ячеек памяти. Второй аргумент задает размер каждой ячейки в байтах. В нашем случае тип long использует 4 байта, поэтому показанный выше оператор устанавливает 100 единиц по 4 байта, задействовав в общей сложности 400 байтов для хранения данных.

Применение sizeof (long) вместо 4 делает код более переносимым. Он будет работать в системах, где тип long имеет размер, отличающийся от 4.

Функция calloc() обладает еще одним свойством: она устанавливает в 0 все биты в блоке. (Однако следует отметить, что в некоторых аппаратных системах значение с плавающей запятой 0 не представляется всеми битами, установленными в 0.)

Функция free() может также использоваться для освобождения памяти, выделенной с помощью calloc().

Динамическое распределение памяти является ключевым средством во многих развитых технологиях программирования. Мы исследуем некоторые из них в главе 17. Вполне вероятно, что ваша библиотека С предлагает ряд других функций управления памятью, при этом часть из них переносима, а часть – нет. Уделите время, чтобы взглянуть на них.

Динамическое распределение памяти и массивы переменной длины

Функциональность массивов переменной длины и malloc() кое в чем пересекается. Например, оба средства могут применяться для создания массива, размер которого определяется во время выполнения:

```
int vlamal()
{
    int n;
    int * pi;
    scanf("%d", &n);
    pi = (int *) malloc (n * sizeof(int));
    int ar[n];   // массив переменной длины
    pi[2] = ar[2] = -5;
    ...
}
```

Одно из отличий между ними заключается в том, что массив переменной длины является автоматической памятью. Следствие использования автоматической памяти состоит в том, что пространство памяти, занимаемое массивом переменной длины, освобождается автоматически, когда поток управления покидает блок, в котором массив определен – в этом случае при завершении функции vlamal(). Таким образом, вы не должны переживать о вызове free(). С другой стороны, доступ к массиву, созданному с применением malloc(), не ограничивается одной функцией.

Например, одна функция может создать массив и возвратить указатель, предоставляя доступ к нему вызывающей функции. Завершив работу с массивом, вызывающая функция может вызвать `free()`. Не будет ошибкой, если при вызове `free()` задать указатель, отличающийся от используемого в вызове `malloc()`; нужно только, чтобы указатели содержали один и тот же адрес. Однако вы не должны пытаться освободить тот же самый блок памяти дважды.

Массивы переменной длины более удобны для организации многомерных массивов. Вы можете создать двумерный массив с применением `malloc()`, но синтаксис будет довольно неуклюжим. Если компилятор не поддерживает средство массивов переменной длины, одна из размерностей должна быть зафиксирована, как в вызовах функции:

```
int n = 5;
int m = 6;
int ar2[n][m];      // массив переменной длины n x m
int (* p2)[6];      // работает до выхода стандарта C99
int (* p3)[m];      // требуется поддержка массивов переменной длины
p2 = (int (*)[6]) malloc(n * 6 * sizeof(int));  // массив n * 6
p3 = (int (*)[m]) malloc(n * m * sizeof(int));  // массив n * m
// предыдущее выражение также требует поддержки массивов переменной длины
ar2[1][2] = p2[1][2] = 12;
```

Полезно взглянуть на объявления указателей. Функция `malloc()` возвращает указатель, так что p2 должен быть указателем подходящего типа. Объявление

```
int (* p2)[6];    // работает до выхода стандарта C99
```

говорит о том, что p2 указывает на массив из шести элементов `int`. Это значит, что `p2[i]` будет интерпретироваться как элемент, состоящий из шести значений `int`, а `p2[i][j]` — это одиночное значение `int`.

Во втором объявлении указателя используется переменная для сообщения размера массива, на который ссылается p3. Это означает, что p3 трактуется как указатель на массив переменной длины, и именно потому данный код не будет работать в рамках стандарта C90.

Классы хранения и динамическое распределение памяти

Вас может интересовать, какова связь между классами хранения и динамическим распределением памяти. Давайте посмотрим на идеализированную модель. Вы можете думать о доступной памяти программы как об имеющей три отдельных области: одна для статических переменных с внешним связыванием, внутренним связыванием и без связывания; одна для автоматических переменных; и одна для динамически выделяемой памяти.

Объем памяти, необходимый для классов со статической продолжительностью хранения, известен на этапе компиляции, и данные, которые хранятся в этой области, доступны на всем протяжении выполнения программы. Каждая переменная этих классов появляется, когда программа запускается, и исчезает при ее завершении.

Однако автоматическая переменная начинает существовать, когда поток управления входит в блок кода, содержащий определение переменной, и исчезает после покидания этого блока. Следовательно, по мере вызова программой функций и их завершения, объем памяти, задействованный под автоматические переменные, возрастает и уменьшается. Такая область памяти обычно реализована в виде стека. Это значит, что новые переменные добавляются в память последовательно, в порядке их создания, а удаляются в обратном порядке, когда исчезают.

Динамически выделяемая память появляется при вызове `malloc()` или родственной ей функции и освобождается при вызове `free()`. Постоянство памяти управляется программистом, а не каким-то набором жестких правил, поэтому блок памяти может быть создан в одной функции и освобожден в другой. По этой причине область памяти, применяемая для динамического распределения памяти, может стать фрагментированной, т.е. неиспользованные участки будут идти вперемешку с активными блоками памяти. Кроме того, использование динамической памяти имеет тенденцию быть более медленным процессом, чем работа со стековой памятью.

Обычно программа применяет разные области памяти для статических объектов, автоматических объектов и динамически выделенных объектов. Сказанное демонстрируется в листинге 12.15.

Листинг 12.15. Программа where.c

```
// where.c -- где что находится в памяти?
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int static_store = 30;
const char * pcg = "Строковый литерал";
int main()
{
    int auto_store = 40;
    char auto_string[] = "Автоматический массив char";
    int * pi;
    char * pcl;

    pi = (int *) malloc(sizeof(int));
    *pi = 35;
    pcl = (char *) malloc(strlen("Динамическая строка") + 1);
    strcpy(pcl, "Динамическая строка");

    printf("        static_store: %d по адресу %p\n", static_store, &static_store);
    printf("          auto_store: %d по адресу %p\n", auto_store, &auto_store);
    printf("                 *pi: %d по адресу %p\n", *pi, pi);
    printf("       %s по адресу %p\n", pcg, pcg);
    printf("%s по адресу %p\n", auto_string, auto_string);
    printf("     %s по адресу %p\n", pcl, pcl);
    printf("       %s по адресу %p\n", "Строка в кавычках", "Строка в кавычках");
    free(pi);
    free(pcl);

    return 0;
}
```

Ниже показан вывод, полученный в одной из систем:

```
         static_store: 30 по адресу 00378000
           auto_store: 40 по адресу 0049FB8C
                  *pi: 35 по адресу 008E9BA0
        Строковый литерал по адресу 00375858
Автоматический массив char по адресу 0049FB74
      Динамическая строка по адресу 008E9BD0
        Строка в кавычках по адресу 00375908
```

Как видите, статические данные, включая строковые литералы, занимают одну область, автоматические данные — вторую область, а динамически выделенные данные — третью область (часто называемую *кучей* или *свободным хранилищем*).

Квалификаторы типов ANSI C

Вам уже известно, что переменная характеризуется типом и классом хранения. В стандарте C90 были добавлены еще два свойства: постоянство и изменчивость. Эти свойства объявляются с помощью ключевых слов const и volatile, которые создают *квалифицированные типы*. В стандарте C99 появился третий квалификатор, restrict, предназначенный для содействия компилятору в оптимизации. Наконец, в C11 добавлен четвертый квалификатор, _Atomic. Стандарт C11 предоставляет дополнительную библиотеку, управляемую stdatomic.h, для поддержки параллельного программирования, и _Atomic является частью этой необязательной поддержки.

Стандарт C99 наделяет квалификаторы типов новым свойством — теперь они идемпотентны. Хотя это звучит подобно жесткому требованию, в действительности это означает лишь то, что один и тот же квалификатор можно указывать в объявлении более одного раза, и избыточные квалификаторы игнорируются:

```
const const const int n = 6;  // то же самое, что и const int n = 6;
```

Это делает приемлемой, например, следующую последовательность:

```
typedef const int zip;
const zip q = 8;
```

Квалификатор типа const

Вы уже встречались со случаями использования const в главах 4 и 10. В качестве напоминания: ключевое слово const в объявлении создает переменную, значение которой не может быть изменено посредством присваивания либо инкрементирования/декрементирования. В случае компилятора, совместимого со стандартом ANSI, код

```
const int nochange;  /* указывает, что m является константой */
nochange = 12;       /* не разрешено                         */
```

приводит к выдаче сообщения об ошибке. Тем не менее, инициализировать переменную const можно. Таким образом, следующий код допустим:

```
const int nochange = 12;  /* все в порядке */
```

Предыдущее объявление делает nochange переменной только для чтения. После того, как она инициализирована, изменять ее нельзя.

Ключевое слово const можно применять, например, для создания массива данных, которые в программе не могут изменяться:

```
const int days1[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Использование const с объявлениями указателей и параметров

Применять ключевое слово const при объявлении простой переменной и массива довольно просто. Указатели в этом смысле сложнее, т.к. необходимо проводить различие между объявлением самого указателя как const и превращением в const значения, на которое он указывает. Объявление

```
const float * pf;  /* pf указывает на константное значение float */
```

создает указатель pf, ссылающийся на значение, которое должно оставаться постоянным. Значение самого pf можно изменять. Например, его можно установить для указания на другое значение const. В противоположность этому, объявление

```
float * const pt;  /* pt является указателем const */
```

говорит о том, что значение самого указателя pt не может быть модифицировано.

Он должен всегда ссылаться на один и то же адрес, однако значение, на которое он указывает, может изменяться. Наконец, объявление

```
const float * const ptr;
```

означает, что ptr должен всегда указывать на одну и ту же ячейку, а значение, хранящееся в этой ячейке, не должно изменяться.

Существует и третье место, куда можно поместить const:

```
float const * pfc;  // то же, что и const float * pfc;
```

Как отражает комментарий, помещение const после имени типа и перед символом * означает, что указатель не может использоваться для изменения значения, на которое он ссылается. Выражаясь кратко, ключевое слово const, находящееся где угодно слева от символа *, делает константными данные, а справа – делает константным сам указатель.

Распространенным использованием этого нового ключевого слова является объявление указателей, которые служат формальными параметрами функций. Например, пусть имеется функция по имени display(), которая отображает содержимое массива. Чтобы применить данную функцию, необходимо передать ей в качестве фактического аргумента имя массива, но имя массива одновременно представляет собой и его адрес. Это позволит функции изменять данные из вызывающей функции. Однако следующий прототип предотвращает такое изменение:

```
void display(const int array[], int limit);
```

В прототипе и в заголовке функции объявление параметра const int array[] аналогично объявлению const int * array, и первое объявление говорит о том, что данные, на которые указывает array, не могут быть изменены.

Этой практики придерживается библиотека ANSI C. Если указатель используется только для предоставления функции доступа к значениям, он объявляется как указатель на квалифицированный посредством const тип. Если указатель применяется для изменения данных в вызывающей функции, то ключевое слово const не используется. Например, объявление функции strcat(), принятое в ANSI C, имеет такой вид:

```
char *strcat(char * restrict s1, const char * restrict s2);
```

Вспомните, что функция strcat() добавляет копию второй строки в конец первой строки. Это приводит к модификации первой строки, но оставляет вторую строку неизмененной. Приведенное объявление отражает сказанное. Вскоре мы возвратимся к роли restrict.

Использование const с глобальными данными

Вспомните, что применение глобальных переменных считается рискованным подходом, поскольку данные открыты для ошибочного изменения любой частью программы. Такой риск исчезает, если данные являются константными, так что вполне разумно снабжать глобальные переменных квалификатором const. Можно иметь переменные const, массивы const и структуры const. (Структуры – это составной тип данных, который обсуждается в следующей главе.)

Однако необходимо соблюдать осторожность при совместном использовании константных данных в разных файлах. Для этого предусмотрены две стратегии. Первая заключается в следовании обычным правилам, применяемым в отношении внешних переменных – использование определяющих объявлений в одном файле и ссылочных объявлений (с ключевым словом extern) в других файлах:

```
/* file1.c -- определение нескольких глобальных констант */
const double PI = 3.14159;
const char * MONTHS[12] =
    {"Январь", "Февраль", "Март", "Апрель", "Май", "Июнь", "Июль",
     "Август", "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"};

/* file2.c -- использование констант, определенных где-то в другом месте */
extern const double PI;
extern const * MONTHS[];
```

Второй подход предполагает помещение констант во включаемый файл. Здесь придется предпринять дополнительное действие, связанное с применением статического внешнего класса хранения:

```
/* constant.h -- определение нескольких глобальных констант */
static const double PI = 3.14159;
static const char * MONTHS[12] =
    {"Январь", "Февраль", "Март", "Апрель", "Май", "Июнь", "Июль",
     "Август", "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"};

/* file1.c -- использование констант, определенных где-то в другом месте */
#include "constant.h"

/* file2.c -- использование констант, определенных где-то в другом месте */
#include "constant.h"
```

Если вы не укажете ключевое слово `static`, то включение заголовочного файла `constant.h` в `file1.c` и `file2.c` приведет к тому, что каждый файл будет иметь определяющее объявление того же самого идентификатора, что стандартом ANSI не поддерживается. (Тем не менее, некоторые компиляторы разрешают это.) Делая идентификатор внешним и статическим, вы фактически предоставляете каждому файлу отдельную копию данных. Такой прием не будет работать, если по замыслу файлы должны использовать эти данные для связи друг с другом, потому что каждый файл будет видеть только свою копию данных. Однако поскольку данные являются константными (из-за наличия ключевого слова `const`) и идентичными (т.к. оба файла включают тот же самый заголовочный файл), проблемы не возникают.

Преимущество подхода с заголовочным файлом состоит в том, что вы не обязаны помнить о применении определяющих объявлений в одном файле и ссылочных объявлений в другом; все файлы просто включают тот же самый заголовочный файл. Недостаток связан с тем, что данные дублируются. В предшествующих примерах это не приводит к значительной проблеме, но она может возникнуть, если в состав константных данных входят крупные массивы.

Квалификатор типа volatile

Квалификатор `volatile` сообщает компилятору, что переменная может иметь значение, которое изменяется действиями, внешними по отношению к программе. Он обычно указывается для аппаратных адресов и для данных, которые совместно используются с другими программами или потоками, выполняющимися одновременно. Например, адрес может ссылаться на текущее показание системных часов. Значение по этому адресу меняется с изменением показаний времени вне зависимости от того, что делает программа. Либо же адрес может применяться для получения информации, переданной, скажем, из другого компьютера.

Синтаксис этого квалификатора подобен синтаксису `const`:

```
volatile int loc1;    /* loc1 является изменчивой ячейкой   */
volatile int * ploc;  /* ploc указывает на изменчивую ячейку */
```

Эти операторы объявляют loc1 как значение volatile и ploc как указатель на значение volatile.

Концепция квалификатора volatile довольно интересна, и вы наверняка хотите узнать, почему комитет ANSI счел необходимым сделать volatile ключевым словом. Причина в том, что оно облегчает проведение оптимизации компилятором. Предположим, например, что есть такой код:

```
val1 = x;
/* код, в котором x не используется */
val2 = x;
```

Интеллектуальный (оптимизирующий) компилятор может заметить, что объект x используется два раза без изменения в промежутке его значения. Он временно может сохранить значение x в регистре. Затем, когда x понадобится для val2, появляется возможность сэкономить время, прочитав значение из регистра, а не из исходной ячейки памяти. Такая процедура называется *кешированием*. Обычно кеширование является полезной оптимизацией, но не в случае, когда значение x изменяется в промежутке между двумя операторами каким-то другим действием. Без ключевого слова volatile у компилятора нет никаких средств, чтобы выяснить, может ли это случиться. Следовательно, во избежание ошибки компилятор не мог реализовать кеширование. Так было до выхода стандарта ANSI. Однако теперь, если в объявлении отсутствует ключевое слово volatile, компилятор может предположить, что значение не изменяется между двумя его применениями, и попытаться оптимизировать данный код.

Значение может быть одновременно и const, и volatile. Например, значение аппаратных часов обычно не должно изменяться программой, что делает его const, но может быть изменено внешним действием, поэтому оно является volatile. Просто поместите оба квалификатора в объявление, как показано ниже; порядок их следования роли не играет:

```
volatile const int loc;
const volatile int * ploc;
```

Квалификатор типа restrict

Ключевое слово restrict расширяет вычислительную поддержку, выдавая компилятору разрешение на оптимизацию определенных разновидностей кода. Оно может быть применено только к указателям и сообщает о том, что тот или иной указатель представляет собой единственное первичное средство доступа к объекту данных. Чтобы понять, почему это полезно, необходимо рассмотреть несколько примеров. Взгляните на показанные ниже объявления:

```
int ar[10];
int * restrict restar = (int *) malloc(10 * sizeof(int));
int * par = ar;
```

Здесь указатель restar является единственным первичным средством доступа в память, выделенную malloc(). Следовательно, он может быть квалифицирован с помощью ключевого слова restrict. Однако указатель par не является ни первичным, ни единственным средством доступа к данным в массиве ar, поэтому он не может быть квалифицирован как restrict.

Теперь рассмотрим несколько искусственный пример, в котором n имеет тип int:

```
for (n = 0; n < 10; n++)
{
   par[n] += 5;
   restar[n] += 5;
```

```
    ar[n] *= 2;
    par[n] += 3;
    restar[n] += 3;
}
```

Зная, что указатель restar – единственное первичное средство доступа к блоку данных, на который он ссылается, компилятор может заменить два оператора, в которых задействован restar, одним оператором, дающим тот же результат:

```
restar[n] += 8;  /* корректная замена */
```

Однако сведение в один двух операторов, в которых участвует par, вызывает вычислительную ошибку:

```
par[n] += 8;      / * дает неправильный ответ */
```

Причина получения неправильного ответа связана с тем, что внутри цикла ar используется для изменения значения данных между двумя случаями доступа к тем же данным с помощью par.

Без ключевого слова restrict компилятор должен рассчитывать на худший случай, а именно – на то, что какой-то другой идентификатор мог изменить данные между двумя применениями указателя. При наличии restrict компилятор получает свободу в поиске вычислительных сокращений.

Ключевое слово restrict можно использовать в качестве квалификатора для параметров функции, которые являются указателями. Это значит, что компилятор может предположить, что внутри тела функции данные, указываемые такими параметрами, не модифицируются с помощью других идентификаторов, и есть возможность попробовать оптимизации, которые иначе бы не предпринимались. Например, библиотека C содержит две функции для копирования байтов из одного места в другое. В стандарте C99 они имеют следующие прототипы:

```
void * memcpy(void * restrict s1, const void * restrict s2, size_t n);
void * memmove(void * s1, const void * s2, size_t n);
```

Каждая функция копирует n байтов из местоположения s2 в местоположение s1. Функция memcpy() требует, чтобы между местоположениями не было перекрытия, но для функции memmove() такое требование отсутствует. Объявление s1 и s2 как restrict означает, что каждый указатель является единственным средством доступа, поэтому они не могут обращаться к одному и тому же блоку данных. Это соответствует требованию отсутствия перекрытия. Функция memmove(), которая разрешает перекрытие, при копировании данных должна соблюдать большую осторожность, чтобы не перезаписать данные до того, как они будут использованы.

Ключевое слово restrict имеет две аудитории. Одна из них – компилятор, и restrict сообщает компилятору о том, что он волен делать определенные предположения, касающиеся оптимизации. Другой аудиторией является пользователь, и restrict говорит пользователю о том, что должны применяться только аргументы, удовлетворяющие требованиям restrict. В общем случае компилятор не способен выяснить, соблюдаете ли вы это ограничение, но вы берете на себя риск за его игнорирование.

Квалификатор типа _Atomic (C11)

Параллельное программирование разделяет выполнение программы на потоки, которые могут выполняться параллельно. При этом возникают сложные задачи программирования, в число которых входит управление разными потоками, получающими доступ к одним и тем же данным.

В C11 предоставлены (необязательные) методы управления, организованные в виде необязательных файлов stdatomic.h и threads.h. Одним из аспектов является концепция атомарного типа, для которого доступ управляется разнообразными функциональными макросами. Пока поток выполняет атомарную операцию над объектом атомарного типа, другие потоки не будут иметь доступа к этому объекту. Например, код наподобие

```
int hogs;   // обычное объявление
hogs = 12;  // обычное присваивание
```

можно было бы заменить следующим кодом:

```
_Atomic int hogs;          // hogs - атомарная переменная
atomic_store(&hogs, 12);   // макрос из stdatomic.h
```

Здесь сохранение значения 12 в hogs представляет собой атомарный процесс, в течение которого другие потоки не будут иметь доступа к hogs.

На время написания этих строк поддержка этой возможности компиляторами только ожидалась.

Новые места для старых ключевых слов

Стандарт C99 позволяет помещать квалификаторы типа и квалификатор класса хранения static в первоначальные квадратные скобки формального параметра в прототипе и заголовке функции. В случае квалификаторов типа это предоставляет альтернативный синтаксис для существующей возможности. Например, вот объявление со старым синтаксисом:

```
void ofmouth(int * const a1, int * restrict a2, int n);  // старый стиль
```

Объявление говорит о том, что a1 — указатель const на int, и это означает, что константным является сам указатель, но не данные, на которые он указывает. Кроме того, объявление отражает то, что a2 представляет собой указатель restrict, как было описано в предыдущем разделе. Эквивалентный новый синтаксис выглядит следующим образом:

```
void ofmouth(int a1[const], int a2[restrict], int n);    // разрешено
                                                         // стандартом C99
```

По существу новое правило разрешает использовать эти два квалификатора либо с формой записи с указателями, либо с формой записи с массивами в объявлении параметров функции.

Случай со static отличается, т.к. он вводит новое и несвязанное применение для этого ключевого слова. Вместо указания области видимости или связывания для переменной со статическим классом хранения новое использование сообщает компилятору, как формальный параметр будет применяться. Например, взгляните на следующий прототип:

```
double stick(double ar[static 20]);
```

Такое использование static отражает то, что фактический аргумент в вызове функции будет указателем на первый элемент массива, имеющего, по меньшей мере, 20 элементов. Цель заключается в том, чтобы позволить компилятору применить эту информацию для оптимизации его кодирования функции. Зачем использовать данное ключевое слово в такой отличающейся манере? Комитет по стандартам C весьма неохотно идет на создание новых ключевых слов, поскольку это может сделать недействительными старые программы, в которых такие слова применялись в качестве идентификаторов, поэтому если удается реализовать новое использование какого-то старого ключевого слова, то так они и поступят.

Как и `restrict`, ключевое слово `static` имеет две аудитории. Одна из них – это компилятор, которому `static` сообщает, что он имеет возможность делать определенные предположения, касающиеся оптимизации. Другой аудиторией является пользователь, которому `static` указывает на необходимость предоставлять только такие аргументы, которые удовлетворяют требованиям `static`.

Ключевые понятия

Язык C предлагает несколько моделей управления памятью. Вы должны ознакомиться со всеми различными вариантами. Вы также должны выработать критерии, когда выбирать тот или иной тип. Большую часть времени наилучшим выбором будет автоматическая переменная. Если вы решите применять другой тип, то для этого должны иметь вескую причину. При взаимодействии между функциями обычно лучше всего использовать автоматические переменные, параметры функций и возвращаемые значения, а не глобальные переменные. С другой стороны, глобальные переменные особенно удобны для представления константных данных.

Вы должны понимать свойства статической памяти, автоматической памяти и выделенной памяти. В частности, имейте в виду, что объем применяемой статической памяти определяется на этапе компиляции, а статические данные загружаются в память при загрузке программы. Память под автоматические переменные выделяется и освобождается во время выполнения, поэтому объем памяти, занимаемой автоматическими переменными, на протяжении выполнения программы меняется. Автоматическую память можно представлять как перезаписываемое рабочее пространство. Выделенная память увеличивается и уменьшается в объеме, но в этом случае процесс управляется вызовами функций, а не происходит автоматически.

Резюме

Память, задействованная под хранение данных в программе, может быть охарактеризована продолжительностью хранения, областью видимости и связыванием. Продолжительность хранения бывает статической, автоматической или выделенной. При статической продолжительности хранения память выделяется в начале выполнения программы и остается занятой на протяжении всего периода выполнения. Если продолжительность хранения является автоматической, то память под переменную выделяется, когда поток управления программы входит в блок, в котором переменная определена, и освобождается, когда управление покидает этот блок. В случае выделенной продолжительности хранения память выделяется вызовом `malloc()` (или родственной функции) и освобождается вызовом `free()`.

Область видимости определяет, какие части программы могут иметь доступ к данным. Переменная, определенная вне любых функций, имеет область видимости в пределах файла и видна любой функции, определенной после объявления этой переменной. Переменная, определенная внутри блока или в качестве параметра функции, имеет область видимости в пределах блока и видна только этому блоку и всем вложенным в него блокам.

Связывание описывает диапазон, в пределах которого переменная, определенная в одной единице программы, может быть привязана к какой-то другой ее единице. Переменные с областью видимости в пределах блока, будучи локальными, не имеют связывания. Переменные с областью видимости в пределах файла, имеют внутреннее или внешнее связывание. Внутреннее связывание означает, что переменная может использоваться только в файле, содержащем ее определение. Внешнее связывание означает, что переменная может применяться также и в других файлах.

Ниже описаны классы хранения в C (кроме концепций, относящихся к потокам).

- **Автоматический.** Переменная, объявленная в блоке (или в качестве параметра в заголовке функции) без модификатора класса хранения или с модификатором класса хранения `auto`, принадлежит к автоматическому классу хранения. Она характеризуется автоматической продолжительностью хранения, областью видимости в пределах блока и отсутствием связывания. Если она не инициализирована, то ее значение не определено.
- **Регистровый.** Переменная, объявленная в блоке (или в виде параметра в заголовке функции) с модификатором класса хранения `register`, принадлежит к регистровому классу хранения. Она характеризуется автоматической продолжительностью хранения, областью видимости в пределах блока и отсутствием связывания. Адрес такой переменной получать нельзя. Объявление переменной как регистровой — это подсказка компилятору о необходимости обеспечить насколько возможно быстрый доступ. Если она не инициализирована, то ее значение не определено.
- **Статический, без связывания.** Переменная, объявленная в блоке с модификатором класса хранения `static`, принадлежит к классу хранения "статический, без связывания". Она характеризуется статической продолжительностью хранения, областью видимости в пределах блока и отсутствием связывания. Такая переменная инициализируется только один раз на этапе компиляции. Если она не инициализирована явно, ее биты устанавливаются в 0.
- **Статический, внешнее связывание.** Переменная, которая определена как внешняя по отношению к любой функции и без использования модификатора класса хранения `static`, принадлежит к классу хранения "статический, внешнее связывание". Она имеет статическую продолжительность хранения, область видимости в пределах файла и внешнее связывание. Такая переменная инициализируется только один раз на этапе компиляции. Если она не инициализирована явно, ее биты устанавливаются в 0.
- **Статический, внутреннее связывание.** Переменная, которая определена как внешняя по отношению к любой функции и с указанием модификатора класса хранения `static`, принадлежит к классу хранения "статический, внутреннее связывание". Она имеет статическую продолжительность хранения, область видимости в пределах файла и внутреннее связывание. Такая переменная инициализируется только один раз на этапе компиляции. Если она не инициализирована явно, ее биты устанавливаются в 0.

Выделение памяти обеспечивается функцией `malloc()` (или родственной ей), которая возвращает указатель на блок памяти, имеющий запрошенное количество байтов. Эту память можно сделать доступной для повторного использования, вызвав функцию `free()` с передачей адреса в качестве аргумента.

Квалификаторами типа являются `const`, `volatile` и `restrict`. Квалификатор `const` указывает на константные данные. В случае применения с указателями `const` может определять, что константным является сам указатель или же данные, на которые указатель ссылается, в зависимости от места его размещения внутри объявления. Квалификатор `volatile` говорит о том, что данные могут изменяться процессами, внешними по отношению к программе. Он предназначен для предупреждения компилятора о том, что он должен избегать оптимизаций, которые предполагались бы в отсутствие `volatile`. Квалификатор `restrict` также введен по причинам, связанным с оптимизацией. Указатель, помеченный с помощью `restrict`, идентифицируется как единственное средство доступа к блоку данных.

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля приведены в приложении А.

1. Какие классы хранения создают переменные, локальные по отношению к функции, которая их содержит?
2. Какие классы хранения создают переменные, которые сохраняются на протяжении выполнения содержащей их программы?
3. Какой класс хранения создает переменные, которые могут использоваться в нескольких файлах? Только в одном файле?
4. Какой вид связывания имеют переменные с областью видимости в пределах блока?
5. Для чего используется ключевое слово `extern`?
6. Взгляните на следующий фрагмент кода:

   ```
   int * pl = (int *) malloc(100 * sizeof(int));
   ```

   Чем от него отличается показанный ниже оператор в смысле конечного результата?

   ```
   int * pl = (int *) calloc(100, sizeof(int));
   ```

7. Каким функциям известна каждая переменная в следующем коде? Если ли в коде ошибки?

   ```
   /* файл 1 */
   int daisy;
   int main(void)
   {
      int lily;
      ...;
   }
   int petal()
   {
      extern int daisy, lily;
      ...;
   }
   /* файл 2 */
   extern int daisy;
   static int lily;
   int rose;
   int stem()
   {
      int rose;
      ...;
   }
   void root()
   {
      ...;
   }
   ```

8. Что выведет следующая программа?

   ```
   #include <stdio.h>
   char color= 'B';
   void first(void);
   void second(void);
   ```

```
int main(void)
{
   extern char color;
   printf("color в main() равно %c\n", color);
   first();
   printf("color в main() равно %c\n", color);
   second();
   printf("color в main() равно %c\n", color);
   return 0;
}
void first(void)
{
   char color;
   color = 'R';
   printf("color в first() равно %c\n", color);
}
void second(void)
{
   color = 'G';
   printf("color в second() равно %c\n", color);
}
```

9. Файл начинается со следующих объявлений:

```
static int plink;
int value_ct(const int arr[], int value, int n);
```

а. Что говорят эти объявления о намерениях программиста?

б. Увеличит ли защиту значений в вызывающей программе замена объявлений `int` и `int n` объявлениями `const int` и `const int n`?

Упражнения по программированию

1. Перепишите программу из листинга 12.4 так, чтобы в ней не использовались глобальные переменные.
2. Расход бензина обычно измеряется в милях на один галлон в США и в литрах на 100 километров в Европе. Ниже приведена часть программы, которая предлагает пользователю выбрать режим (метрический или американский), а затем выполняет сбор данных и вычисляет расход топлива:

```
// pe12-2b.c
// компилировать вместе с pe12-2a.c
#include <stdio.h>
#include "pe12-2a.h"
int main(void)
{
  int mode;
  printf("Введите 0 для метрического режима или 1 для американского режима: ");
  scanf("%d", &mode);
  while (mode >= 0)
  {
     set_mode(mode);
     get_info();
     show_info();
     printf("Введите 0 для метрического режима или 1 для американского режима");
     printf(" (-1 для завершения): ");
     scanf("%d", &mode);
  }
```

```
  printf("Программа завершена.\n");
  return 0;
}
```

Ниже показан пример вывода:

```
Введите 0 для метрического режима или 1 для американского режима: 0
Введите пройденное расстояние в километрах: 600
Введите объем израсходованного топлива в литрах: 78.8
Расход топлива составляет 13.13 литров на 100 км.
Введите 0 для метрического режима или 1 для американского режима
(-1 для завершения): 1
Введите пройденное расстояние в милях: 434
Введите объем израсходованного топлива в галлонах: 12.7
Расход топлива составляет 34.2 мили на галлон.
Введите 0 для метрического режима или 1 для американского режима
(-1 для завершения): 3
Указан недопустимый режим. Используется режим 1 (американский).
Введите пройденное расстояние в милях: 388
Введите объем израсходованного топлива в галлонах: 15.3
Расход топлива составляет 25.4 мили на галлон.
Введите 0 для метрического режима или 1 для американского режима
(-1 для завершения): -1
Программа завершена.
```

Если пользователь введет некорректный режим, программа сообщает об этом и использует режим, который был выбран в последний раз. Реализуйте заголовочный файл pe12-2a.h и файл исходного кода pe12-2a.c, чтобы обеспечить работоспособность программе. В файле исходного кода должны определяться три переменных с областью видимости в пределах файла и внутренним связыванием. Одна переменная представляет режим, вторая – расстояние и третья – расход топлива. Функция get_info() запрашивает ввод данных согласно выбранному режиму и сохраняет ответы в переменных с областью видимости в пределах файла. Функция show_info() вычисляет и отображает расход топлива на основе выбранного режима. Можете считать, что пользователь вводит только числовые значения.

3. Переделайте программу, описанную в упражнении 2, чтобы в ней использовались только автоматические переменные. Обеспечьте в программе тот же самый пользовательский интерфейс, т.е. она должна предлагать пользователю ввести режим и т.д. Однако у вас будет другой набор вызовов функций.

4. Напишите и протестируйте в цикле функцию, которая возвращает количество ее вызовов.

5. Напишите программу, которая генерирует список из 100 случайных чисел в диапазоне от 1 до 10, отсортированный по убыванию. (Можете приспособить к типу int алгоритм сортировки из главы 11, только в этом случае сортируйте сами числа.)

6. Напишите программу, которая генерирует 1000 случайных чисел в диапазоне от 1 до 10. Не сохраняйте и не выводите эти числа, а выводите лишь то, сколько раз генерировалось каждое число. Программа должна делать это для 10 разных начальных значений. Появляются ли числа в одинаковых количествах? Можете использовать функции из этой главы или функции rand() и srand() из ANSI C, которые следуют тому же формату, что и рассмотренные здесь. Это один из способов исследования недетерминированности конкретного генератора случайных чисел.

7. Напишите программу, которая ведет себя подобно модификации листинга 12.13, которая обсуждалась после представления вывода из программы, показанной в листинге 12.13. То есть программа должна давать вывод следующего вида:

```
Введите количество бросаний или q для завершения: 18
Сколько граней и сколько костей? 6 3
Имеем 18 бросаний 3 костей с 6 гранями.
  12  10   6   9   8  14   8  15   9  14  12  17  11   7  10
  13   8  14
Введите количество бросаний или q для завершения: q
```

8. Ниже приведена часть программы:

```
// pe12-8.c
#include <stdio.h>
int * make_array(int elem, int val);
void show_array(const int ar[], int n);
int main(void)
{
    int * pa;
    int size;
    int value;

    printf("Введите количество элементов: ");
    while (scanf("%d", &size) == 1 && size > 0)
    {
        printf("Введите значение для инициализации: ");
        scanf("%d", &value);
        pa = make_array(size, value);
        if (pa)
        {
            show_array(pa, size);
            free(pa);
        }
        printf("Введите количество элементов (<1 для завершения): ");
    }
    printf("Программа завершена.\n");

    return 0;
}
```

Завершите программу, предоставив определения функций `make_array()` и `show_array()`. Функция `make_array()` принимает два аргумента. Первый аргумент – это количество элементов в массиве значений `int`, а второй аргумент – значение, которое должно быть присвоено каждому элементу массива. Эта функция использует `malloc()` для создания массива подходящего размера, присваивает каждому элементу заданное значение и возвращает указатель на массив. Функция `show_array()` отображает содержимое массива по восемь элементов в строке.

9. Напишите программу со следующим поведением. Сначала она запрашивает количество слов, которые нужно ввести. Затем она предлагает ввести слова, после чего их отображает. Воспользуйтесь `malloc()` и ответом на первый запрос (количество слов), чтобы создать динамический массив с подходящим количеством указателей на `char`. (Обратите внимание, что поскольку каждый элемент в массиве является указателем на `char`, возвращаемое значение функции `malloc()` должно сохраняться в указателе на указатель на `char`.) При чтении строки программа должна читать слово во временный массив элементов `char`, с помощью

malloc() выделять пространство, достаточное для хранения слова, и помещать адрес в массив указателей на char. Далее программа должна копировать слово из временного массива в выделенное пространство памяти. Таким образом, в итоге получается массив указателей на char, каждый из которых ссылается на объект с размером, необходимым для хранения конкретного слова. Результаты пробного запуска должны выглядеть следующим образом:

```
Сколько слов вы хотите ввести? 5
Теперь введите 5 слов:
Мне понравилось выполнять это упражнение
Вот введенные вами слова:
Мне
понравилось
выполнять
это
упражнение
```

13

Файловый ввод-вывод

В ЭТОЙ ГЛАВЕ...

- Функции: fopen(), getc(), putc(), exit(), fclose(), fprintf(),fscant(), fgets(),fputs(), rewind(), fseek(),ftell(),fflush(), fgetpos(), fsetpos(),feof(),terror(), ungetc(), setvbuf(), tread(), fwrite()
- Обработка файлов с использованием семейства стандартных функций ввода-вывода С
- Текстовые и двоичные режимы, текстовый и двоичный форматы, буферизированный и небуферизированный ввод-вывод
- Применение функций, которые позволяют осуществлять последовательный и произвольный доступ в файлы

Файлы являются неотъемлемой частью современных компьютерных систем. Они используются для хранения программ, документов, данных, корреспонденции, форм, изображений, фотографий, музыкальных произведений, видеоклипов и несметного числа других видов информации. Как программист, вы должны писать программы, которые создают файлы, записывают в файлы и читают из файлов. В этой главе мы покажем, как это делать.

Взаимодействие с файлами

Часто вам нужны программы, которые могут читать информацию из файлов или записывать результаты в файл. В главе 8 вы узнали, что одной из таких форм взаимодействия между программой и файлом является перенаправление в файл. Этот метод прост, но ограничен. Например, пусть необходимо написать интерактивную программу, которая запрашивает названия книг, а затем сохраняет весь список в файле. Если вы примените перенаправление, как в

```
books > bklist
```

то все интерактивные запросы будут перенаправлены в `bklist`. Это не только приведет к помещению в файл `bklist` нежелательного текста, но и предотвратит отображение вопросов, на которые вы предположительно ответили.

Как и можно было ожидать, язык C предлагает более мощные методы взаимодействия с файлами. Он позволяет открывать файл внутри программы и затем с помощью специальных функций ввода-вывода осуществлять чтение и запись в этот файл. Однако прежде чем переходить к изучению таких методов, давайте кратко рассмотрим саму природу файла.

Понятие файла

Файл — это именованный раздел хранилища, обычно расположенный на жестком диске или на получающем все большее распространение в последнее время твердотельном диске. Например, `stdio.h` можно представлять себе как имя файла, содержащего некоторую полезную информацию. Тем не менее, для операционной системы файл выглядит не так просто. Скажем, крупный файл может храниться в нескольких отдельных фрагментах или содержать дополнительные данные, которые позволяют операционной системе определять вид этого файла. Но за все это отвечает операционная система, а не вы (если только вы сами не занимаетесь разработкой операционной системы). Вас интересует только то, как файлы представляются в программе на C.

В языке C файл рассматривается как непрерывную последовательность байтов, каждый из которых может быть прочитан индивидуально. Это соответствует файловой структуре в среде Unix, откуда C берет свое начало. Поскольку другие среды могут не соответствовать в точности этой модели, в C предлагаются два способа представления файлов: текстовый режим и двоичный режим.

Текстовый режим и двоичный режим

Прежде всего, давайте проведем различие между текстовым и двоичным содержимым, текстовым и двоичным файловыми форматами, а также текстовым и двоичным режимами для файлов.

Содержимое всех файлов хранится в двоичной форме (нули и единицы). Но если в файле двоичные коды символов (к примеру, ASCII или Unicode) используются главным образом для представления текста, почти как в строках C, то такой файл являет-

ся текстовым, т.е. имеет текстовое содержимое. Если же двоичные значения в файле представляют код на машинном языке, числовые данные (с применением того же внутреннего представления, как, скажем, у значений `long` или `double`), кодировку изображения или музыкального произведения, то содержимое будет двоичным.

Для обоих видов содержимого в Unix используется один и тот же файловый формат. Учитывая, что язык C был создан как инструмент для разработки операционной системы Unix, не должно вызывать удивления, что и в C, и в Unix для обозначения разрыва строки внутри текста применяется `\n` (символ перевода строки). Каталоги Unix поддерживают счетчик размера файла, который программы могут использовать для выяснения, достигнут ли конец файла. Тем не менее, другие системы располагали другими способами поддержки файлов, специально ориентированными на хранение текста. Это значит, что в них для текстовых файлов предусмотрен формат, отличающийся от модели Unix.

Например, в файлах Macintosh до выхода OS X для указания новой строки применялся символ `\r` (возврат каретки). В файлах из ранних версий MS-DOS для обозначения новой строки использовалась комбинация `\r\n`, а для признака конца файла – встраиваемый символ <Ctrl+Z>, несмотря на то, что действительный файл может быть дополнен нулевыми символами, чтобы сделать общий размер кратным 256. (В среде Windows редактор "Блокнот" по-прежнему создает текстовые файлы в формате MS-DOS, но более новые редакторы могут применять формат, близкий к Unix-подобному.) Другие системы могут делать все строки в текстовом файле одинаковой длины, дополняя их при необходимости нулевыми символами до нужной длины. Или же система может кодировать длину каждой строки в ее начале.

Чтобы привнести некоторую закономерность в обработку текстовых файлов, язык C предоставляет два способа доступа в файл: *двоичный* режим и *текстовый* режим. В двоичном режиме программе доступен каждый байт файла. Однако в текстовом режиме то, что видит программа, может отличаться от того, что хранится в файле. В текстовом режиме при чтении файла представление локальной среды для таких символов, как конец строки или конец файла, сопоставляется с их представлением в C.

Аналогично, представление в C отображается на локальное представление вывода. Например, программа C, скомпилированная в старой среде Macintosh, будет преобразовывать `\r` в `\n` при чтении файла в текстовом режиме и `\n` в `\r` при записи в файл. Программа C текстового режима, скомпилированная на платформе MS-DOS, будет преобразовывать `\r\n` в `\n` при чтении из файла и `\n` в `\r\n` при записи в файл. Программы текстового режима, написанные для других сред, предпринимают похожие корректировки.

При работе с текстовым файлом вы не ограничены только текстовым представлением. Для того же самого файла можно использовать и двоичное представление. Если вы поступите подобным образом для старого текстового файла MS-DOS, то программа будет видеть в файле символы `\r` и `\n`; никакого сопоставления не происходит. (Сказанное иллюстрируется на рис. 13.1.) Если вы хотите написать программу просмотра текста, которая работает, скажем, со старыми форматами Macintosh, MS-DOS и Unix/Linux, то могли бы применять двоичный режим, чтобы программа выясняла фактическое содержимое файла и предпринимала соответствующие действия.

Несмотря на то что в C доступны двоичное и текстовое представления, они могут быть реализованы идентично. Как упоминалось ранее, поскольку в Unix применяется всего одна файловая структура, в реализациях для Unix оба представления одинаковы. То же самое справедливо для Linux.



Рис. 13.1. Двоичное и текстовое представления

Уровни ввода-вывода

В дополнение к выбору представления файла в большинстве случаев вы можете выбирать один из двух уровней ввода-вывода (т.е. из двух уровней управления доступом к файлам). *Низкоуровневый ввод-вывод* предусматривает использование основных служб ввода-вывода, предоставляемых операционной системой. *Стандартный высокоуровневый ввод-вывод* предполагает применение стандартного пакета библиотечных функций C и определений из заголовочного файла `stdio.h`. Стандарт C поддерживает только стандартный пакет ввода-вывода, т.к. нет никакой возможности гарантировать, что все операционные системы могут быть представлены одинаковой низкоуровневой моделью ввода-вывода. Отдельные реализации могут также предлагать низкоуровневые библиотеки, но из-за того, что стандарт C устанавливает переносимую модель ввода-вывода, мы сосредоточим все внимание на ней.

Стандартные файлы

Программы на C автоматически открывают три файла, которые называются *стандартным вводом, стандартным выводом* и *стандартным выводом ошибок*. По умолчанию стандартный ввод представляет собой обычное устройство ввода в вашей системе, как правило, клавиатуру. Стандартный вывод и стандартный вывод ошибок по умолчанию являются обычным устройством вывода вашей системы, т.е. экраном монитора.

Естественно, стандартный ввод обеспечивает ввод данных в программу. Это файл, который читается с помощью функций `getchar()` и `scanf()`. Стандартный вывод — место, куда направляется обычный вывод программы. Он используется функциями `putchar()`, `puts()` и `printf()`. Перенаправление, как вы уже знаете из главы 8, приводит к тому, что другие файлы опознаются как стандартный ввод и стандартный вывод. Назначение файла стандартного вывода ошибок заключается в том, чтобы предоставить логически обособленное место для отправки сообщений об ошибках.

Например, если вместо экрана вы перенаправите вывод в файл, то сообщения, отправляемые в стандартный вывод ошибок, по-прежнему будут попадать на экран. Это удобно, т.к. если бы сообщения об ошибках направлялись также в файл, то вы бы их не увидели до тех пор, пока не просмотрели файл.

Стандартный ввод-вывод

По сравнению с низкоуровневым вводом-выводом стандартный пакет ввода-вывода, помимо переносимости, обладает еще двумя преимуществами. Во-первых, в нем доступно множество специализированных функций, которые упрощают решение разнообразных задач, связанных с вводом-выводом. Например, функция `printf()` преобразует различные формы данных в строковый вывод, подходящий для терминалов. Во-вторых, ввод и вывод являются *буферизированными*. Это значит, что информация передается крупными порциями (обычно по 512 и более байтов), а не по одному байту за раз. Например, когда программа читает файл, порция данных считывается в буфер — промежуточную область памяти. Такая буферизации существенно увеличивает скорость передачи данных. Затем программа может исследовать отдельные байты в этом буфере. Буферизация происходит "за кулисами", поэтому создается иллюзия посимвольного доступа. (Вы также можете буферизировать низкоуровневый ввода-вывода, но большую часть работы придется проделать самостоятельно.)

В листинге 13.1 показано, как применять стандартный ввод-вывод для чтения файла и подсчета количества находящихся в нем символов. Свойства программы из листинга 13.1 мы обсудим в нескольких последующих разделах. (Эта программа использует аргументы командной строки. Если вы работаете в среде Windows, можете после компиляции запустить программу в окне командной строки. Если вы имеете дело с Macintosh, проще всего скомпилировать и запустить программу в форме командной строки с использованием Terminal. Или же, как объяснялось в главе 11, в XCode посредством меню Products (Продукты) можно предоставить аргументы командной строки для программы, запускаемой из IDE-среды. В качестве альтернативы программу можно изменить так, чтобы для получения имени файла вместо аргумента командной строки применялись функции `puts()` и `gets()`.)

Листинг 13.1. Программа count.c

```
/* count.c -- использование стандартного ввода-вывода */
#include <stdio.h>
#include <stdlib.h> // прототип exit()

int main(int argc, char *argv[])
{
   int ch;          // место для хранения каждого символа по мере чтения
   FILE *fp;        // "указатель файла"
   unsigned long count = 0;
   if (argc != 2)
   {
      printf("Использование: %s имя_файла\n", argv[0]);
      exit(EXIT_FAILURE);
   }
   if ((fp = fopen(argv[1], "r")) == NULL)
   {
      printf("Не удается открыть %s\n", argv[1]);
      exit(EXIT_FAILURE);
   }
```

```
    while ((ch = getc(fp)) != EOF)
    {
       putc(ch,stdout);  // то же, что и putchar(ch);
       count++;
    }
    fclose(fp);
    printf("Файл %s содержит %lu символов\n", argv[1], count);

    return 0;
}
```

Проверка наличия аргумента командной строки

Первым делом программа в листинге 13.1 проверяет значение argc, чтобы выяснить, имеется ли аргумент командной строки. Если аргумент отсутствует, программа выводит инструкцию по ее использованию и завершается. Строка argv[0] — это имя программы. Явное применение argv[0] вместо имени программы обеспечит автоматическое изменение сообщения об ошибке, если вы измените имя исполняемого файла. Такая возможность полезна в средах, подобных Unix, которые разрешают иметь много имен для одного файла. Но будьте осторожны — некоторые операционные системы могут не распознавать argv[0], поэтому такой прием не является полностью переносимым.

Функция exit() приводит к прекращению работы программы с закрытием всех открытых файлов. Аргумент exit() передается некоторым операционным системам, включая Unix, Linux, Windows и MS-DOS, где он может использоваться другими программами. Обычное соглашение предусматривает возвращение 0 при успешном завершении и ненулевого значения в случае ненормального завершения.

Разные выходные значения могут применяться для обозначения различных причин отказа, и это установившаяся практика программирования в средах Unix и DOS. Тем не менее, не все операционные системы распознают один и тот же диапазон возможных возвращаемых значений. Вследствие этого стандарт С устанавливает использование довольно ограниченного минимального диапазона. В частности, стандарт требует, чтобы при успешном завершении программы применялось значение 0 или макрос EXIT_SUCCESS, а неудачное завершение указывалось с помощью макроса EXIT_FAILURE. Эти макросы, наряду с прототипом exit(), находятся в заголовочном файле stdlib.h.

В рамках ANSI С использование return в первоначальном вызове main() дает тот же результат, что и вызов exit(). По этой причине в main() оператор

```
return 0;
```

который встречался на протяжении всей книги, эквивалентен следующему оператору:

```
exit(0);
```

Однако обратите внимание на определяющую фразу "в первоначальном вызове". Если функция main() задействована в рекурсивной программе, то exit() по-прежнему приводит к прекращению ее работы, но return передает управление предыдущему уровню рекурсии до тех пор, пока не будет достигнут первоначальный уровень. Только после этого return завершает выполнение программы. Еще одно отличие между return и exit() состоит в том, что функция exit() прекращает работу программы, даже если она вызвана в функции, отличной от main().

Функция fopen()

Далее в программе с помощью функции fopen() открывается файл. Эта функция объявлена в заголовочном файле stdio.h. Ее первым аргументом является имя файла, который необходимо открыть; точнее, это адрес строки, содержащей имя файла. Второй аргумент – строка, идентифицирующая режим, в котором файл должен быть открыт. В библиотеке C предоставляется несколько возможностей, показанных в табл. 13.1.

Таблица 13.1. Строки режима для fopen()

Строка режима	Описание
"r"	Открыть текстовый файл для чтения
"w"	Открыть текстовый файл для записи с усечением существующего файла до нулевой длины или созданием файла, если он не существует
"a"	Открыть текстовый файл для записи с добавлением данных в конец существующего файла или созданием файла, если он не существует
"r+"	Открыть текстовый файл для обновления (т.е. для чтения и записи)
"w+"	Открыть текстовый файл для обновления (чтения и записи), предварительно выполнив усечение файла до нулевой длины, если он существует, или создав файл, если его нет
"a+"	Открыть текстовый файл для обновления (чтения и записи) с добавлением данных в конец существующего файла или созданием файла, если он не существует; читать можно весь файл, но записывать допускается только в конец файла
"rb", "wb", "ab", ab+", "a+b", "wb+", "w+b", "ab+", "a+b"	Подобны предыдущим режимам, за исключением того, что вместо текстового режима они используют двоичный режим
"wx", "wbx", "w+x", "wb+x" или "w+bx"	(C11) Подобны режимам без буквы x, за исключением того, что они отказываются работать, если файл существует, и открывают файл в монопольном режиме, если это возможно

Для таких систем, как Unix и Linux, которые имеют только один файловый тип, режимы с буквой b эквивалентны соответствующим режимам без буквы b.

Новые режимы записи C11 с буквой x обладают парой новых характеристик по сравнению со старыми режимами записи. Во-первых, если вы пытаетесь открыть существующий файл в одном из традиционных режимов записи, то fopen() усекает файл до нулевой длины, в результате чего предыдущее содержимое файла утрачивается. Но режимы с буквой x обеспечивают в таком случае отказ функции fopen() от дальнейшей работы, не причиняя вреда файлу. Во-вторых, при условии, что это позволяет среда, возможность монопольного доступа в режимах с x предотвращает доступ к файлу со стороны других программ или потоков до тех пор, пока текущий процесс не закроет этот файл.

Внимание!

Если вы используете любой режим "w" без буквы x для существующего файла, содержимое файла усекается так, что программа может начать работу с чистого листа. Однако попытка открыть существующий файл с применением одного из режимов C11, содержащий букву x, завершится отказом.

После успешного открытия файла функция fopen() возвращает *указатель файла*, который затем другие функции ввода-вывода могут использовать для указания этого файла. Указатель файла (в примере это fp) имеет тип указателя на FILE; здесь FILE – производный тип, определенный в stdio.h. Указатель fp не ссылается на действительный файл. Вместо этого он указывает на объект данных, содержащий информацию о файле, включая сведения о буфере, который применяется для файлового ввода-вывода. Так как функции ввода-вывода из стандартной библиотеки используют буфер, им необходимо знать, где этот буфер находится. Им также должно быть известно, насколько заполнен буфер и с каким файлом осуществляется работа. Это позволяет функциям по мере необходимости заполнять или опустошать буфер. Вся эта информация содержится в объекте данных, указываемом fp. (Такой объект данных является примером структуры C, которые обсуждаются в главе 14.)

Функция fopen() возвращает нулевой указатель (также определенный в stdio.h), если ей не удается открыть файл. Когда указатель fp равен NULL, программа прекращает выполнение. Функция fopen() может отказать из-за переполнения диска, отсутствия файла в искомом каталоге, недопустимого имени, ограничений доступа или аппаратной проблемы. Это лишь небольшая часть причин отказа, так что ищите неполадку; даже минимальные меры по отлавливанию ошибок могут иметь большое значение.

Функции getc() и putc()

Функции getc() и putc() работают очень похоже на getchar() и putchar(). Отличие заключается в том, что этим новым функциям потребуется указать, с каким файлом работать. Таким образом, приведенный ниже многократно использованный нами оператор означает "получить символ из стандартного ввода":

```
ch = getchar();
```

Тем не менее, следующий оператор означает "получить символ из файла, идентифицируемого fp":

```
ch = getc(fp);
```

Аналогично, показанный далее оператор означает "поместить символ ch в файл, идентифицируемый указателем fpout на FILE":

```
putc(ch, fpout);
```

В списке аргументов putc() сначала задается символ, а затем указатель файла.

В листинге 13.1 во втором аргументе putc() применяется stdout. Он определен в stdio.h как указатель файла, ассоциированный со стандартным выводом, поэтому putc(ch,stdout) эквивалентно putchar(ch). На самом деле вторая функция обычно определена как первая. Аналогично, getchar() определена как функция getc(), использующая стандартный ввод.

Возможно, вас интересует, почему в этом примере применяется putc(), а не putchar(). Одна причина связана с необходимостью ознакомления с функцией putc(). Другая причина заключается в том, что вы легко можете преобразовать программу так, чтобы она могла генерировать файловый за счет использования аргумента, отличного от stdout.

Конец файла

Программа, читающая данные из файла, должна останавливаться, когда она достигает конца файла. Как можно сообщить программе о том, что встретился конец файла?

Функция getc() возвращает специальное значение EOF, если она пытается прочитать символ и обнаруживает, что достигнут конец файла. Таким образом, программа C выясняет, что она достигла конца файла, только после попытки чтения за концом файла. (Это не похоже на поведение в ряде других языков, в которых предусмотрена специальная функция для проверки на предмет конца файла *перед* попыткой чтения.)

Чтобы избежать проблем с попыткой чтения пустого файла, при файловом вводе должен применяться цикл с входным условием (не цикл do while). Из-за конструктивных особенностей getc() (и других функций ввода C) программа должна выполнять чтение до входа в тело цикла. Тогда показанное ниже проектное решение вполне подойдет:

```
// правильное проектное решение #1
int ch;              // переменная int для хранения EOF
FILE * fp;
fp = fopen("wacky.txt", "r");
ch = getc(fp);       // получить первоначальный ввод
while (ch != EOF)
{
   putchar(ch);      // обработать ввод
   ch = getc(fp);    // получить следующий ввод
}
```

Это решение можно ужать следующим образом:

```
// правильное проектное решение #2
int ch;
FILE * fp;
fp = fopen("wacky.txt", "r");
while (( ch = getc(fp)) != EOF)
{
   putchar(ch);      // обработать ввод
}
```

Поскольку оператор ввода является частью проверочного условия while, он выполняется до того, как поток управления войдет в тело цикла.

Проектных решений вроде приведенного ниже вы должны избегать:

```
// неудачное проектное решение (две проблемы)
int ch;
FILE * fp;
fp = fopen("wacky.txt", "r");
while (ch != EOF)    // первым используется неопределенное значение ch
{
   ch = getc(fp);    // получить ввод
   putchar(ch);      // обработать ввод
}
```

Первая проблема связана с тем, что когда переменная ch в первый раз сравнивается с EOF, ей еще не было присвоено значение. Вторая проблема в том, что если getc() возвращает EOF, то цикл пытается обработать EOF, как если бы это был допустимый символ. Указанные дефекты поддаются исправлению. К примеру, вы могли бы инициализировать ch каким-то фиктивным значением и поместить внутрь цикла оператор if, но зачем об этом беспокоиться, если доступны правильные проектные решения?

Указанные меры предосторожности касаются и других функций ввода. Они также возвращают сигнал об ошибке (EOF или пустой указатель), столкнувшись с концом файла.

Функция fclose()

Функция fclose(fp) закрывает файл, идентифицируемый fp, при необходимости сбрасывая буферы. В более ответственной программе вы должны удостовериться, что файл закрыт успешно. Функция fclose() возвращает значение 0, если файл был закрыт успешно, и EOF, если нет:

```
if (fclose(fp) != 0)
   printf("Ошибка при закрытии файла %s\n", argv[1]);
```

Функция fclose() может завершиться неудачно, если, например, жесткий диск заполнен, съемное устройство хранения извлечено или произошла ошибка ввода-вывода.

Указатели на стандартные файлы

В stdio.h три указателя файлов ассоциированы с тремя стандартными файлами, которые автоматически открываются программами на С.

Стандартный файл	Указатель файла	Обычное устройство
Стандартный ввод	stdin	Клавиатура
Стандартный вывод	stdout	Экран
Стандартный вывод ошибок	stderr	Экран

Все они имеют тип указателя на FILE, поэтому могут использоваться в качестве аргументов для стандартных функций ввода-вывода подобно fp в приведенном ранее примере. Давайте теперь перейдем к рассмотрению примера, в котором создается новый файл и в него производится запись.

Бесхитростная программа уплотнения файла

Следующая программа копирует избранные данные из одного файла в другой. Она открывает два файла одновременно с применением режима "r" для одного и режима "w" для второго. Программа (показанная в листинге 13.2) уплотняет содержимое первого файла, грубо оставляя только каждый третий символ. В итоге она помещает уплотненный текст во второй файл. Имя второго файла образуется путем дополнения старого имени расширением .red. Использование аргументов командной строки, открытие одновременно более одного файла и добавление расширения к имени файла в общем случае являются довольно практичными приемами. Эта конкретная форма уплотнения файла имеет ограниченное применение, но, как вы увидите, такие случаи возникают. (Программу несложно модифицировать, чтобы для предоставления имен файлов вместо аргументов командной строки использовались стандартные методы ввода-вывода.)

Листинг 13.2. Программа reducto.c

```
// reducto.c -- сокращение файлов на две трети!
#include <stdio.h>
#include <stdlib.h>    // для exit()
#include <string.h>

int main(int argc, char *argv[])
{
    FILE  *in, *out;   // объявление двух указателей на FILE
    int ch;
```

```
    char name[LEN];     // хранилище для имени выходного файла
    int count = 0;

    // проверка аргументов командной строки
    if (argc < 2)
    {
       fprintf(stderr, "Использование: %s имя_файла\n", argv[0]);
       exit(EXIT_FAILURE);
    }
    // настройка ввода
    if ((in = fopen(argv[1], "r")) == NULL)
    {
       fprintf(stderr, "Не удается открыть файл \"%s\"\n",
              argv[1]);
       exit(EXIT_FAILURE);
    }
    // настройка вывода
    strncpy(name,argv[1], LEN - 5); // копирование имени файла
    name[LEN - 5] = '\0';
    strcat(name,".red");             // добавление .red
    if ((out = fopen(name, "w")) == NULL)
    {                                // открытие файла для записи
       fprintf(stderr, "Не удается создать выходной файл.\n");
       exit(3);
    }
    // копирование данных
    while ((ch = getc(in)) != EOF)
       if (count++ % 3 == 0)
          putc(ch, out);             // выводить каждый третий символ
    // очистка
    if (fclose(in) != 0 || fclose(out) != 0)
       fprintf(stderr, "Ошибка при закрытии файлов.\n");

    return 0;
}
```

Предположим, что исполняемый файл называется reducto, и мы применяем его к файлу по имени Eddy, который содержит единственную строку:

```
So even Eddy came oven ready.
```

Команда имеет такой вид:

```
reducto eddy
```

Вывод записывались в файл по имени eddy.red. Программа ничего не выводит на экран, но отображение содержимого eddy.red должно выявить следующее:

```
Send money
```

Этот пример иллюстрирует несколько приемов программирования. Давайте рассмотрим некоторые из них.

Функция fprintf() подобна printf() за исключением того, что она требует передачи в первом аргументе указателя файла. Мы использовали указатель stderr для отправки сообщений об ошибках в стандартный вывод ошибок; это стандартная практика в C.

Чтобы сконструировать новое имя для выходного файла, в программе применяется функция strncpy() для копирования имени eddy в массив name. Аргумент LEN - 5 оставляет место для суффикса .red и завершающего нулевого символа. Нулевой сим-

вол не копируется, если длина строки argv[2] больше LEN - 5, поэтому на всякий случай добавляется нулевой символ. После вызова strncpy() первый нулевой символ в name перезаписывается символом точки из .red, когда функция strcat() добавляет эту строку, давая в результате eddy.red. Кроме того, в программе предусмотрена проверка, удалось ли открыть файл с именем eddy.red. Это особенно важно в определенных средах, где имя файла наподобие strange.c.red может быть недопустимым. Например, в традиционной системе DOS нельзя добавлять расширение к расширению. (Подход, подходящий для MS-DOS, предполагает замену любого существующего расширения вариантом .red, так что уплотненной версией strange.c будет strange.red. Можно было бы воспользоваться функцией strchr(), чтобы найти точку в имени, если она есть, и копировать только часть строки до точки.)

В программе имеются два одновременно открытых файла, из-за чего объявлены два указателя на FILE. Обратите внимание, что файлы открываются и закрываются независимо друг от друга. Существует предельное количество одновременно открытых файлов, которое зависит от системы и реализации; часто этот предел находится в диапазоне от 10 до 20. Один и тот же указатель файла можно использовать для разных файлов при условии, что эти файлы не открываются в одно и то же время.

Файловый ввод-вывод: fprintf(), fscanf(), fgets() и fputs()

Для каждой функции ввода-вывода из предшествующих глав имеется похожая функция файлового ввода-вывода. Главное отличие между ними заключается в том, что функциям файлового ввода-вывода с помощью указателя на FILE необходимо сообщать, с каким файлом работать. Подобно getc() и putc(), эти функции требуют идентификации файла с применением указателя на FILE, такого как stdout, либо использования возвращаемого значения fopen().

Функции fprintf() и fscanf()

Функции файлового ввода-вывода fprintf() и fscanf() работают аналогично printf() и scanf(), отличаясь только наличием дополнительного первого аргумента, в котором идентифицируется подходящий файл. Вы уже применяли функцию fprintf(). В листинге 13.3 демонстрируется работа функций файлового ввода-вывода наряду с функцией rewind().

Листинг 13.3. Программа addaword.c

```
/* addaword.c -- использование fprintf(), fscanf() и rewind() */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 41

int main(void)
{
   FILE *fp;
   char words[MAX];

   if ((fp = fopen("wordy", "a+")) == NULL)
   {
      fprintf(stdout, " Не удается открыть файл \"wordy\".\n");
      exit(EXIT_FAILURE);
   }
```

```
    puts("Введите слова для добавления в файл; для завершения");
    puts("введите символ # в начале строки.");
    while ((fscanf(stdin,"%40s", words) == 1)  && (words[0] != '#'))
        fprintf(fp, "%s\n", words);

    puts("Содержимое файла:");
    rewind(fp);         /* возврат в начало файла */
    while (fscanf(fp,"%s",words) == 1)
        puts(words);
    puts("Готово!");
    if (fclose(fp) != 0)
        fprintf(stderr, "Ошибка при закрытии файла\n");

    return 0;
}
```

Эта программа позволяет добавлять слова в файл. За счет использования режима "a+" программа может осуществлять чтение и запись в файл. При первом запуске она создает файл wordy и позволяет помещать в него слова по одному в строке. При последующем запуске программа позволяет добавлять (дописывать) слова к существующему содержимому. Режим добавления разрешает только дописывать данные в конец файла, но режим "a+" позволяет читать весь файл. Функция rewind() обеспечивает перемещение в начало файла, так что финальный цикл while может вывести содержимое файла. Обратите внимание, что rewind() принимает указатель файла в качестве аргумента.

Ниже приведены результаты пробного запуска в среде Unix (исполняемый файл был переименован в addaword):

```
$ addaword
Введите слова для добавления в файл; для завершения
введите символ # в начале строки.
Один талантливый программист
#
Содержимое файла:
Один
талантливый
программист
Готово!
$ addaword
Введите слова для добавления в файл; для завершения
введите символ # в начале строки.
сумел многих
удивить
#
Содержимое файла:
Один
талантливый
программист
сумел
многих
удивить
Готово!
```

Как видите, fprintf() и fscanf() работают подобно функциям printf() и scanf(). В отличие от putc(), функции fprintf() и fscanf() принимают указатель на FILE в первом, а не последнем аргументе.

Функции fgets() и fputs()

Вы уже сталкивались с fgets() в главе 11. Как и в отброшенной функции gets(), ее первым аргументом является адрес (типа char *), где должны сохраняться введенные данные. Второй аргумент – целое число, представляющее максимальный размер входной строки. Заключительный аргумент – это указатель файла, который идентифицирует файл, подлежащий чтению. Вызов функции выглядит следующим образом:

```
fgets(buf, STLEN, fp);
```

Здесь buf – это имя массива char, MAX – максимальный размер строки, а fp – указатель на FILE.

Как вы уже знаете, функция fgets() читает входные данные до появления первого символа новой строки – до тех пор, пока не будет прочитано количество символов, на единицу меньше верхнего предела, либо пока не будет обнаружен конец файла; затем fgets() добавляет завершающий нулевой символ, чтобы сформировать строку. Таким образом, верхний предел представляет максимальное количество символов плюс нулевой символ. Если fgets() удастся прочитать целую строку до достижения предельного числа символов, она поместит символ новой строки непосредственно перед нулевым символом, отметив конец строки. Функция fgets() возвращает значение NULL, когда сталкивается с EOF. Этим можно воспользоваться для проверки признака конца файла. В противном случае она возвращает переданный ей адрес.

Функция fputs() принимает два аргумента: адрес строки и указатель файла. Она записывает строку, находящуюся в указанной ячейке, в заданный файл. В отличие от puts(), функция fputs() при выводе не добавляет символ новой строки. Вызов fputs() выглядит следующим образом:

```
fputs(buf, fp);
```

Здесь buf является адресом строки, а fp идентифицирует целевой файл.

Поскольку fgets() сохраняет символ новой строки, а fputs() не добавляет этот символ, они хорошо работают в тандеме. Как было показано в листинге 11.8, они успешно действуют вместе, даже когда STLEN меньше длины входной строки.

Произвольный доступ: fseek() и ftell()

Функция fseek() позволяет трактовать файл подобно массиву и переходить непосредственно к любому байту в файле, открытом с помощью fopen(). Чтобы ознакомиться с работой fseek(), давайте напишем программу (листинг 13.4), которая отображает содержимое файла в обратном порядке. Обратите внимание, что fseek() принимает три аргумента и возвращает значение int. Функция ftell() возвращает текущую позицию в файле как значение long.

Листинг 13.4. Программа reverse.c

```
/* reverse.c -- отображение содержимого файла в обратном порядке */
#include <stdio.h>
#include <stdlib.h>
#define CNTL_Z '\032'   /* маркер конца файла в текстовых файлах DOS */
#define SLEN 81
int main(void)
{
   char file[SLEN];
```

```
    char ch;
    FILE *fp;
    long count, last;

    puts("Введите имя файла для обработки:");
    scanf("%80s", file);
    if ((fp = fopen(file,"rb")) == NULL)
    {                                        /* режим только для чтения      */
       printf("reverse не удается открыть %s\n", file);
       exit(EXIT_FAILURE);
    }
    fseek(fp, 0L, SEEK_END);                 /* перейти в конец файла        */
    last = ftell(fp);
    for (count = 1L; count <= last; count++)
    {
       fseek(fp, -count, SEEK_END);          /* двигаться в обратном направлении */
       ch = getc(fp);
       if (ch != CNTL_Z && ch != '\r')       /* файлы MS-DOS */
          putchar(ch);
    }
    putchar('\n');
    fclose(fp);

    return 0;
}
```

Вот вывод программы для одного из файлов:

```
Введите имя файла для обработки:
Cluv
.С екызя ан ьтавориммаргорп ястиварн енм огесв ешьлоБ
```

В этой программе применяется двоичный режим, поэтому она может иметь дело как с текстовыми файлами MS-DOS, так и с файлами Unix. Однако ее работа может оказаться некорректной в среде, в которой для текстовых файлов используется какой-то другой формат.

На заметку!

Если вы запускаете программу в среде командной строки, то программа ожидает, что файл с указанным именем находится в том же каталоге (или папке), что и сама исполняемая программа. Если программа запускается из IDE-среды, то каталог, в котором производится поиск файла, зависит от реализации. Например, по умолчанию Microsoft Visual Studio 2012 просматривает каталог, содержащий исходный код, а XCode 4.6 ищет файл в каталоге, где расположен исполняемый файл.

Теперь нам нужно обсудить три темы: как работают функции `fseek()` и `ftell()`, каким образом применяется двоичный поток данных и как делать программу переносимой.

Работа функций `fseek()` и `ftell()`

Первым из трех аргументов функции `fseek()` является указатель `FILE` на файл, в котором будет производиться поиск. Файл должен быть открыт с помощью `fopen()`.

Второй аргумент `fseek()` называется *смещением*. Он показывает, насколько далеко необходимо переместиться от стартовой точки (ниже приведен список режимов стартовых точек). В этом аргументе должно передаваться значение `long`, которое может

быть положительным (переместиться вперед), отрицательным (переместиться назад) или нулевым (остаться на месте).

Третий аргумент устанавливает режим, идентифицирующий стартовую точку. Начиная со стандарта ANSI, в заголовочном файле `stdio.h` указаны следующие именованные константы для режимов:

Режим	Откуда измеряется смещение
SEEK_SET	От начала файла
SEEK_CUR	От текущей позиции
SEEK_END	От конца файла

В более старых реализациях такие определения могут отсутствовать и вместо них для указания режимов используются числовые значения `0L`, `1L` и `2L`, соответственно. Вспомните, что суффикс `L` идентифицирует значения тип `long`. Или же в реализации могут быть предусмотрены константы, определенные в другом заголовочном файле. В случае сомнений обращайтесь к руководству пользователя или онлайновому справочнику.

Ниже приведены примеры вызова функции (`fp` – указатель файла):

```
fseek(fp, 0L, SEEK_SET);      // перейти в начало файла
fseek(fp, 10L, SEEK_SET);     // перейти на 10 байтов от начала файла
fseek(fp, 2L, SEEK_CUR);      // перейти вперед на 2 байта от текущей позиции
fseek(fp, 0L, SEEK_END);      // перейти в конец файла
fseek(fp, -10L, SEEK_END);    // перейти назад на 10 байтов от конца файла
```

С такими вызовами связаны возможные ограничения, о которых речь пойдет позже в главе.

Значение, возвращаемое `fseek()`, равно 0, если все в порядке, и -1, если возникла ошибка вроде попытки выхода за границы файла.

Функция `ftell()` имеет тип `long` и возвращает текущую позицию в файле. В стандарте ANSI C она объявлена в `stdio.h`. Поскольку изначально функция `ftell()` была реализована в Unix, она указывает позицию в файле, возвращая количество байтов от начала файла, причем первый байт получает номер 0, второй – номер 1 и т.д. В ANSI C такое определение применяется к файлам, открытым в двоичном режиме, но не обязательно к файлам, открытым в текстовом режиме. Это одна из причин использования двоичного режима в листинге 13.5.

Теперь мы можем исследовать базовые элементы программы из листинга 13.5. Прежде всего, оператор

```
fseek(fp, 0L, SEEK_END);
```

устанавливает позицию со смещением 0 байтов от конца файла. Это означает установку позиции в конец файла. Затем оператор

```
last = ftell(fp);
```

присваивает `last` количество байтов от начала до конца файла.

Далее следует цикл:

```
for (count = 1L; count <= last; count++)
{
   fseek(fp, -count, SEEK_END);    /* двигаться в обратном направлении */
      ch = getc(fp);
}
```

На первой итерации происходит позиционирование на первый символ перед концом файла (т.е. на финальный символ в файле). После этого данный символ выводится. На следующей итерации позиция устанавливается на предпоследний символ в файле, который затем выводится. Процесс продолжается до тех пор, пока не будет достигнут и выведен первый символ в файле.

Сравнение двоичного и текстового режимов

Программа в листинге 13.4 спроектирована так, чтобы работать в средах Unix и MS-DOS. В Unix имеется только один файловый формат, поэтому никакие специальные корректировки не нужны. Однако MS-DOS требует дополнительного внимания. Многие редакторы MS-DOS помечают конец текстового файла посредством символа <Ctrl+Z>. Когда такой файл открывается в текстовом режиме, в C этот символ распознается как признак конца файла. Тем не менее, когда тот же самый файл открывается в двоичном режиме, <Ctrl+Z> является обычным символом в файле, а действительный признак конца файла появляется позже. Он может находиться сразу после <Ctrl+Z> или же файл может быть дополнен нулевыми символами, чтобы сделать размер файла кратным, скажем, 256. Нулевые символы в среде MS-DOS не отображаются, поэтому мы предусмотрели код, предотвращающий вывод символа <Ctrl+Z>.

О другом отличии мы упоминали ранее: символ новой строки текстового файла в MS-DOS представлен с помощью комбинации `\r\n`. Программа на C, открывающая тот же самый файл в текстовом режиме, "видит" символы `\r\n` как просто `\n`, но в случае применения двоичного режима программа видит оба символа, т.е. `\r` и `\n`. По данной причине был включен код для подавления вывода `\r`. Поскольку текстовый файл в Unix обычно не содержит ни <Ctrl+Z>, ни `\r`, этот дополнительный код не затрагивает большинство текстовых файлов в Unix.

Функция `ftell()` может работать по-разному в текстовом и двоичном режимах. Форматы текстовых файлов многих систем заметно отличаются от модели Unix, в которых подсчет байтов от начала файла не дает осмысленной величины. В стандарте ANSI C утверждается, что для случая текстового режима `ftell()` возвращает значение, которое может быть использовано в качестве второго аргумента `fseek()`. Например, в MS-DOS функция `ftell()` может возвращать количество, при подсчете которого комбинация `\r\n` рассматривается как один байт.

Переносимость

В идеальном случае функции `fseek()` и `ftell()` должны соответствовать модели Unix. Тем не менее, отличия в реальных системах иногда делают это невозможным. По этой причине стандарт ANSI предоставляет для этих функций пониженные ожидания. Далее описаны некоторые ограничения.

- В двоичном режиме реализации не обязаны поддерживать режим `SEEK_END`. Поэтому переносимость кода из листинга 13.4 не гарантируется. Более переносимый подход предусматривает чтение всего файла байт за байтом, пока не встретится конец. Но последовательное чтение файла для нахождения конца медленнее, чем просто переход в его конец. Директивы условной компиляции препроцессора C, обсуждаемые в главе 16, предлагают более систематизированный способ для поддержки выбора альтернативного кода.

- В текстовом режиме будут гарантированно работать только следующие вызовы `fseek()`:

Вызов функции	Результат
`fseek(file, 0L, SEEK_SET)`	Перейти в начало файла
`fseek(file, 0L, SEEK_CUR)`	Оставаться в текущей позиции
`fseek(file, 0L, SEEK_END)`	Перейти в конец файла
`fseek(file, ftell_pos, SEEK_SET)`	Перейти в позицию `ftell_pos` от начала файла; `ftell_pos` — это значение, возвращаемое функцией `ftell()`

К счастью, многие распространенные среды делают возможными более строгие реализации этих функций.

Функции fgetpos() и fsetpos()

Одна потенциальная проблема с функциями `fseek()` и `ftell()` заключается в том, что они ограничивают размеры файлов значениями, которые могут быть представлены типом `long`. Возможно, 2 миллиарда байтов могут показаться более чем достаточным размером, но постоянно растущие объемы устройств хранения позволяют работать с файлами больших размеров. В ANSI С появились две новых функции позиционирования, которые спроектированы на работу с файлами крупных размеров. Вместо применения для представления позиции значения `long`, они используют новый тип под названием `fpos_t` (от file position type — тип для позиции в файле). Тип `fpos_t` не является фундаментальным, а определяется в терминах других типов. Переменная или объект данных типа `fpos_t` может указывать позицию внутри файла и не может быть массивом, но его природа подобного и не требует. Реализация может предоставить какой-то тип, удовлетворяющий нуждам конкретной платформы; такой тип может быть реализован, например, в виде структуры.

В ANSI С определено, как применять тип `fpos_t`. Функция `fgetpos()` имеет следующий прототип:

```
int fgetpos(FILE * restrict stream, fpos_t * restrict pos);
```

Вызов `fgetpos()` помещает текущее значение типа `fpos_t` в ячейку, указанную `pos`; это значение описывает позицию в файле. Функция возвращает ноль в случае успеха и ненулевое значение при отказе.

Прототип функции `fsetpos()` выглядит так:

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

Вызов `fsetpos()` приводит к использованию значения типа `fpos_t` из ячейки, заданной с помощью `pos`, для установки указателя файла в позицию, которую отражает это значение. Функция возвращает ноль в случае успеха и ненулевое значение при отказе. Значение `fpos_t` должно было быть получено предыдущим вызовом `fgetpos()`.

"За кулисами" стандартного ввода-вывода

Теперь, когда вы увидели некоторые возможности стандартного пакета ввода-вывода, мы займемся исследованием репрезентативной концептуальной модели, чтобы посмотреть, каким образом работает стандартный ввод-вывод.

Обычно первым шагом в применении стандартного ввода-вывода является вызов функции `fopen()` для открытия файла. (Однако вспомните, что файлы `stdin`, `stdout` и `stderr` открываются автоматически.)

Функция fopen() не только открывает файл, но и настраивает буфер (или два буфера для режимов чтения-записи) и устанавливает структуру данных, содержащую сведения о файле и о буфере. Кроме того, fopen() возвращает указатель на эту структуру, так что другие функции знают, где ее искать. Предположим, что это значение присвоено переменной типа указателя по имени fp. Говорят, что функция fopen() "открывает поток данных". Если файл открывается в текстовом режиме, вы получаете текстовый поток, а если в двоичном – то двоичный поток.

Эта структура данных обычно включает индикатор позиции в файле, предназначенный для определения текущей позиции в потоке. Она также содержит индикаторы для ошибок и конца файла, указатель на начало буфера, идентификатор файла и счетчик количества байтов, действительно скопированных в буфер.

Давайте сосредоточим внимание на файловом вводе. Обычно следующим шагом является вызов одной из функций ввода, объявленных в stdio.h, таких как fscanf(), getc() или fgets(). Вызов любой такой функции приводит к тому, что порция данных копируется из файла в буфер. Размер буфера зависит от реализации, но обычно он имеет 512 или кратное этому числу количество байтов, такое как 4 096 или 16 384. (По мере увеличения объемов жестких дисков и памяти компьютера, выбираемые размеры буферов также имеют тенденцию к росту.) Вдобавок к заполнению буфера первоначальный вызов функции устанавливает значения в структуре, указываемой посредством fp. В частности, устанавливается текущая позиция в потоке данных и количество байтов, скопированных в буфер. Обычно текущая позиция начинается с байта 0.

После инициализации структуры данных и буфера функция ввода читает запрошенные данные из буфера. В результате индикатор позиции устанавливается так, чтобы указывать на символ, следующий за последним прочитанным символом. Поскольку все функции ввода из семейства stdio.h используют тот же самый буфер, вызов любой такой функции возобновляет чтение там, где оно было остановлено предыдущим вызовом любой из функций.

Когда функция ввода обнаруживает, что все символы из буфера прочитаны, она запрашивает копирование из файла в буфер следующей порции данных с объемом, равным размеру буфера. В такой манере функции ввода могут читать все содержимое вплоть до конца файла. После того, как функция прочитает последний символ финальной порции данных, она устанавливает индикатор конца файла в истинное значение. Следующий вызов любой функции ввода возвратит EOF.

Функции вывода в похожем стиле производят запись в буфер. Когда буфер заполняется, данные копируются в файл.

Другие стандартные функции ввода-вывода

Стандартная библиотека ANSI содержит более трех десятков функций, образующих семейство для стандартного ввода-вывода. Мы не будем раскрывать их все, но кратко опишем еще несколько функций, чтобы дать более четкое представление о том, что вообще доступно. Для каждой функции будет приведен прототип С, отражающий ее аргументы и возвращаемое значение. Все обсуждаемые здесь функции кроме setvbuf() также доступны в реализациях, предшествующих ANSI. В разделе V приложения Б представлен полный список пакета стандартного ввода-вывода ANSI С.

Функция intungetc(int c, FILE *fp)

Функция ungetc() заталкивает символ, указанный в с, обратно во входной поток. В случае заталкивания символа во входной поток он будет прочитан следующим вызовом стандартной функции ввода (рис. 13.2).

Предположим, например, что вам нужна функция, которая читает все символы до следующего двоеточия, не включая его. Вы можете применить `getchar()` или `getc()` для чтения символов до двоеточия и затем вызвать `ungetc()`, чтобы вернуть двоеточие обратно во входной поток. Стандарт ANSI C гарантирует только одно заталкивание за один раз. Если реализация разрешает заталкивать сразу несколько символов в строке, функции ввода прочитают их в порядке, обратном заталкиванию.

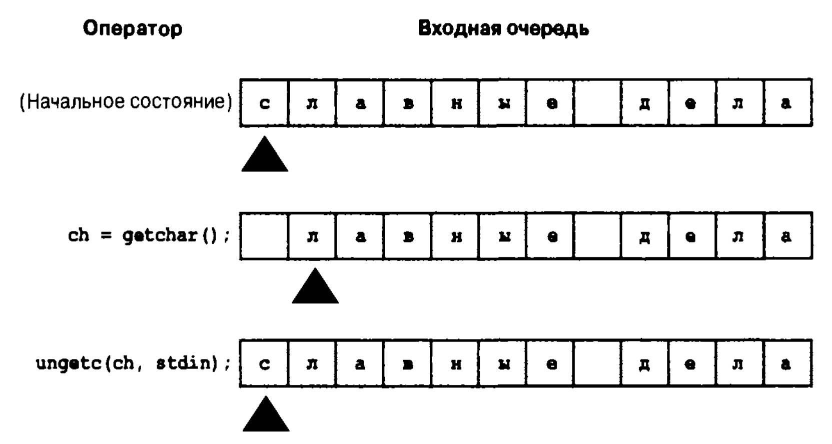


Рис. 13.2. Функция `ungetc()`

Функция `int fflush()`

Прототип `fflush()` выглядит так:

```
int fflush(FILE *fp);
```

Вызов функции `fflush()` приводит к тому, что любые незаписанные данные в буфере вывода отправляются в выходной файл, идентифицируемый с помощью `fp`. Этот процесс называется *сбросом буфера*. Если `fp` — нулевой указатель, то сбрасываются все буферы вывода. Результат использования функции `fflush()` на входном потоке не определен. Ее можно применять с потоком обновления (для любого режима чтения-записи), при условии, что самая последняя операция, использующая поток, не была операцией ввода.

Функция `int setvbuf()`

Прототип `setvbuf()` имеет следующий вид:

```
int setvbuf(FILE * restrict fp, char * restrict buf, int mode, size_t size);
```

Функция `setvbuf()` устанавливает альтернативный буфер, предназначенный для применения стандартными функциями ввода-вывода. Она вызывается после того, как файл был открыт, и перед выполнением любой другой операции на потоке данных. Указатель `fp` идентифицирует поток, а `buf` указывает на используемое хранилище. Значение `buf`, не равное `NULL`, говорит о том, что буфер вы создаете самостоятельно. Например, вы могли бы объявить массив из 1024 элементов `char` и передать адрес этого массива. Однако если в качестве значения `buf` указывается `NULL`, то функция сама выделит память под буфер. Аргумент `size` сообщает `setvbuf()` размер этого массива. (`size_t` — это производный целочисленный тип, который рассматривался в главе 5.) Для `mode` доступны следующие варианты: `_IOFBF` означает полную буферизацию (буфер сбрасывается, когда полон), `_IOLBF` — построчную буферизацию (буфер

сбрасывается, когда полон или когда в него записан символ новой строки) и _IONBF — отсутствие буферизации. Функция возвращает ноль при успешном завершении и ненулевое значение в противном случае.

Предположим, что у вас есть программа, которая работает с сохраненными объектами данных, имеющими размер, скажем, по 3 000 байтов каждый. Вы могли бы с помощью setvbuf() создать буфер, размер которого кратен размеру объекта данных.

Двоичный ввод-вывод: fread() и fwrite()

Следующими в списке идут функции fread() и fwrite(), но сначала мы затронем некоторые основы. Стандартные функции ввода-вывода, которые вы применяли до сих пор, были ориентированы на текст, работая с символами и строками. А что, если в файле нужно сохранить числовые данные? Действительно, можно воспользоваться функцией fprintf() и форматом %f, чтобы сохранить значение с плавающей запятой, но тогда оно сохранится как последовательность символов. Например, код

```
double num = 1./3.;
fprintf(fp,"%f", num);
```

сохраняет num в виде последовательности из восьми символов: 0.333333.

Применение спецификатора %.2f позволяет сохранить его как последовательность из четырех символов: 0.33. Использование спецификатора %.12f дает возможность сохранить его в виде 14 символов: 0.333333333333. Смена спецификаторов приводит к изменению размера пространства, необходимого для значения. После того, как значение num было сохранено как 0.33, нет никакой возможности вернуться к полной точности значения при его чтении из файла. В общем случае функция fprintf() преобразует числовые значения в символьные данные, возможно изменяя значения.

Наиболее точный и единообразный способ сохранения числа предусматривает использование того же самого набора битов, что и компьютер. Таким образом, значение double должно быть сохранено в области с размером как у типа double. Когда данные хранятся в файле в представлении, которое применяется в программе, мы говорим, что данные сохранены в *двоичной форме*. Никакие преобразования из числовых форм в последовательности символов не производятся. Для стандартного ввода-вывода такую услугу предлагают функции fread() и fwrite(), работа которых иллюстрируется на рис. 13.3.

В действительности, как вы помните, все данные хранятся в двоичной форме. Даже символы хранятся с использованием двоичного представления их кодов. Однако если все данные в файле интерпретируются как коды символов, мы говорим, что файл содержит текстовые данные. Если некоторые или все данные интерпретируются как числовые данные в двоичной форме, мы говорим, что файл содержит двоичные данные. (Кроме того, двоичными также являются файлы, в которых данные представляют собой команды на машинном языке.)

Применение терминов *двоичный* и *текстовый* может привести к путанице. Стандарт ANSI C распознает два режима открытия файлов: двоичный и текстовый. Многие операционные системы распознают два файловых формата: двоичный и текстовый. Все эти характеристики связаны, но не идентичны. Вы можете открывать файл текстового формата в двоичном режиме. Вы можете сохранять текст в файле двоичного формата. Вы можете использовать функцию getc() для копирования файлов, содержащих двоичные данные. Однако для сохранения двоичных данных в файле двоичного формата вы обычно будете применять двоичный режим. Аналогично, чаще всего работа с текстовыми данными в текстовых файлах производится при их открытии в текстовом режиме. (Файлы, генерируемые текстовыми процессорами, как правило, являются двоичными, поскольку они содержат много нетекстовой информации, которая описывает шрифты и форматирование.)

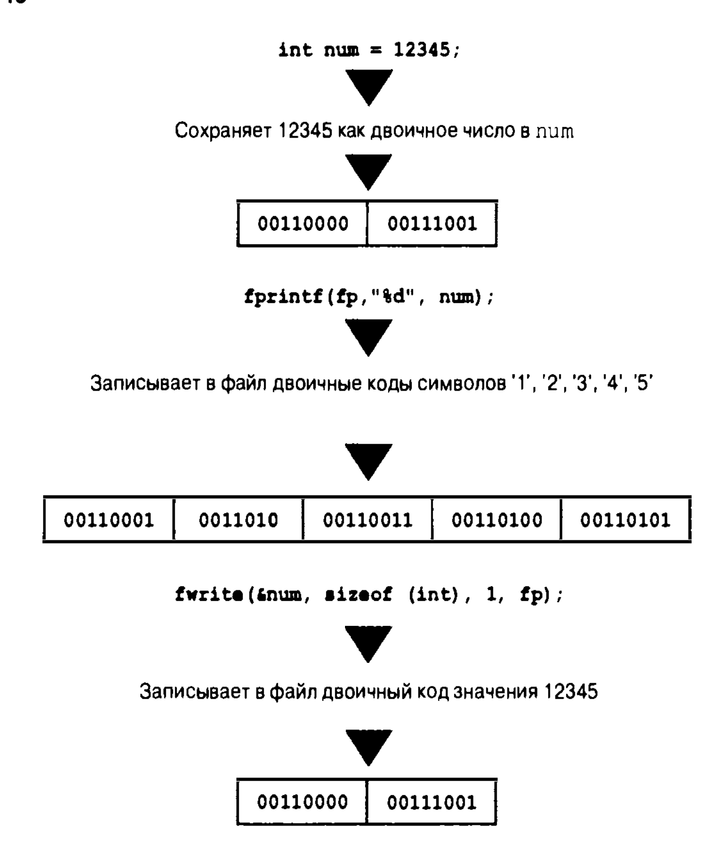


Рис. 13.3. Двоичный и текстовый вывод

Функция size_t fwrite()

Ниже показан прототип функции fwrite():

```
size_t fwrite(const void * restrict ptr, size_t size, size_t nmemb,
              FILE * restrict fp);
```

Функция fwrite() записывает двоичные данные в файл. Тип size_t определен в терминах стандартных типов С. Это тип, возвращаемый операцией sizeof. Обычно им является unsigned int, но реализации могут выбирать другой тип. Указатель ptr – это адрес порции данных, предназначенной для записи. Аргумент size представляет размер в байтах порции данных, подлежащих записи, а nmemb – количество таких порций. Как обычно, fp идентифицирует файл, в который должна производиться запись. Например, чтобы сохранить объект данных (такой как массив) размером 256 байтов, можно поступить так:

```
char buffer[256];
fwrite(buffer, 256, 1, fp);
```

Этот вызов fwrite() записывает одну порцию данных размером 256 байтов из буфера в файл. Чтобы сохранить, скажем, массив из 10 элементов double, понадобятся следующие операторы:

```
double earnings[10];
fwrite(earnings, sizeof (double), 10, fp);
```

Этот вызов fwrite() записывает данные из массива earnings в файл 10 порциями данных, каждая из которых имеет размер double.

Возможно, вы обратили внимание на странное объявление const void * restrict ptr в прототипе fwrite(). Проблема, связанная с функцией fwrite(), заключается в том, что ее первый аргумент не имеет фиксированного типа. Скажем, в первом примере использовался аргумент buffer, имеющий тип указателя на char, а во втором примере – аргумент earnings с типом указателя на double. В контексте прототипов ANSI C эти фактические аргументы преобразуются в тип указателя на void, который действует как своего рода универсальный тип для указателей. (До выхода ANSI C для этого аргумента применялся тип char *, требующий приведения к нему актуальных аргументов.)

Функция fwrite() возвращает количество успешно записанных элементов. Обычно оно равно nmemb, однако может быть меньше, если произошла ошибка записи.

Функция size_t fread()

Прототип функции fread() имеет следующий вид:

```
size_t fread(void * restrict ptr, size_t size, size_t nmemb,
             FILE * restrict fp);
```

Функция fread() принимает такой же набор аргументов, как и fwrite(). На этот раз ptr представляет собой адрес области памяти, куда помещаются данные, прочитанные из файла, а fp идентифицирует читаемый файл. Эту функцию следует использовать для чтения данных, которые были записаны в файл с помощью fwrite(). Например, вот как восстановить массив из 10 элементов double, сохраненный в предыдущем примере:

```
double earnings[10];
fread(earnings, sizeof (double), 10, fp);
```

Этот вызов копирует 10 значений размера double в массив earnings.

Функция fread() возвращает количество успешно прочитанных элементов. Обычно оно равно nmemb, однако может быть меньше, если произошла ошибка записи или был достигнут конец файла.

Функции int feof(FILE *fp) и int ferror(FILE *fp)

Когда стандартные функции ввода возвращают EOF, это обычно означает, что достигнут конец файла. Тем не менее, возврат EOF может также указывать на возникновение ошибки чтения. Функции feof() и ferror() позволяют проводить различие между этими двумя возможностями. Функция feof() возвращает ненулевое значение, если при последнем вызове функции ввода был обнаружен конец файла, и ноль в противном случае. Функция ferror() возвращает ненулевое значение, если произошла ошибка чтения или записи, и ноль в противном случае.

Пример использования fread() и fwrite()

Давайте воспользуемся некоторыми из этих функций в программе, которая добавляет содержимое из списка файлов в конец указанного файла. Одна из задач заключается в передаче внутрь программы информации о файлах. Это можно делать интерактивно или с помощью аргументов командной строки. Мы примем первый подход, который предполагает выполнение перечисленных ниже действий.

- Запрос имени файла назначения и его открытие.
- Применение цикла для запроса исходных файлов.
- Поочередное открытие каждого исходного файла в режиме чтения и добавление его содержимого в конец файла назначения.

Чтобы проиллюстрировать работу функции `setvbuf()`, мы применим ее для установки другого размера буфера. Следующий этап детализации связан с открытием файла назначения. Мы будем использовать следующие шаги.

1. Открытие файла назначения в режиме добавления.
2. Если сделать это не удается, то завершение работы.
3. Установка буфера размером 4096 байтов для этого файла.
4. Если сделать это не удается, то завершение работы.

Аналогично, мы можем уточнить часть программы, отвечающую за копирование, для чего выполнить с каждым файлом такие действия.

- Если это файл назначения, то пропустить его и перейти к следующему файлу.
- Если файл не может быть открыт в режиме чтения, то пропустить его и перейти к следующему файлу.
- Добавить содержимое файла в файл назначения.

В завершение программа перейдет в начало файла назначения и отобразит его содержимое. В целях практики для копирования будут применяться функции `fread()` и `fwrite()`. Результирующий код приведен в листинге 13.5.

Листинг 13.5. Программа append.c

```
/* append.c -- добавление содержимого файлов в файл назначения */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFSIZE 4096
#define SLEN 81
void append(FILE *source, FILE *dest);
char * s_gets(char * st, int n);
int main(void)
{
    FILE *fa, *fs;           // fa для файла назначения, fs для исходного файла
    int files = 0;           // количество добавляемых файлов
    char file_app[SLEN];     // имя файла назначения
    char file_src[SLEN];     // имя исходного файла
    int ch;
    puts("Введите имя файла назначения:");
    s_gets(file_app, SLEN);
    if ((fa = fopen(file_app, "a+")) == NULL)
    {
        fprintf(stderr, "Не удается открыть %s\n", file_app);
        exit(EXIT_FAILURE);
    }
    if (setvbuf(fa, NULL, _IOFBF, BUFSIZE) != 0)
    {
        fputs("Не удается создать выходной буфер\n", stderr);
        exit(EXIT_FAILURE);
    }
```

```
    puts("Введите имя первого исходного файла (или пустую строку для завершения):");
    while (s_gets(file_src, SLEN) && file_src[0] != '\0')
    {
        if (strcmp(file_src, file_app) == 0)
            fputs("Добавить файл в конец самого себя невозможно\n",stderr);
        else if ((fs = fopen(file_src, "r")) == NULL)
            fprintf(stderr, "Не удается открыть %s\n", file_src);
        else
        {
            if (setvbuf(fs, NULL, _IOFBF, BUFSIZE) != 0)
            {
                fputs("Не удается создать входной буфер\n",stderr);
                continue;
            }
            append(fs, fa);
            if (ferror(fs) != 0)
                fprintf(stderr,"Ошибка при чтении файла %s.\n",
                        file_src);
            if (ferror(fa) != 0)
                fprintf(stderr,"Ошибка при записи файла %s.\n",
                        file_app);
            fclose(fs);
            files++;
            printf("Содержимое файла %s добавлено.\n", file_src);
            puts("Введите имя следующего файла (или пустую строку для завершения):");
        }
    }
    printf("Добавление завершено. Количество добавленных файлов: %d.\n", files);
    rewind(fa);
    printf("Содержимое %s:\n", file_app);
    while ((ch = getc(fa)) != EOF)
        putchar(ch);
    puts("Отображение завершено.");
    fclose(fa);
    return 0;
}
void append(FILE *source, FILE *dest)
{
    size_t bytes;
    static char temp[BUFSIZE];      // выделить память один раз
    while ((bytes = fread(temp,sizeof(char),BUFSIZE,source)) > 0)
        fwrite(temp, sizeof (char), bytes, dest);
}
char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;
    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');     // поиск символа новой строки
        if (find)                    // если адрес не является NULL,
            *find = '\0';            // поместить туда нулевой символ
        else
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

Если функции setvbuf() не удается создать буфер, она возвращает ненулевое значение, после чего программа прекращает работу. Похожий код устанавливает буфер размером 4096 байтов для файла, копируемого в текущий момент. За счет использования NULL во втором аргументе setvbuf() мы позволяем этой функции самостоятельно выделить память под буфер.

Для получения имени файла в программе применяется функция s_gets() вместо scanf(), т.к. scanf() пропускает пробельные символы и, следовательно, не сможет обнаружить пустую строку. Кроме того, в программе используется s_gets() вместо простой функции fgets(), потому что fgets() оставляет в строке символ новой строки.

Показанный ниже код предотвращает добавление содержимого файла в конец самого себя:

```
if (strcmp(file_src, file_app) == 0)
   fputs("Добавить файл в конец самого себя невозможно\n",stderr);
```

Аргумент file_app представляет имя файла назначения, а file_src — имя файла, обрабатываемого в текущий момент.

Функция append() выполняет копирование. Вместо копирования по одному байту за раз она применяет fread() и fwrite() для копирования по 4096 байтов за один раз:

```
void append(FILE *source, FILE *dest)
{
   size_t bytes;
   static char temp[BUFSIZE];  // выделить память один раз

   while ((bytes = fread(temp,sizeof(char),BUFSIZE,source)) > 0)
      fwrite(temp, sizeof (char), bytes, dest);
}
```

Поскольку файл, указанный посредством dest, открыт в режиме добавления, содержимое исходных файлов по очереди добавляется в конец файла dest. Обратите внимание, что массив temp имеет статическую продолжительность хранения (это значит, что память под него выделяется на этапе компиляции, а не каждый раз, когда вызывается append()) и область видимости в пределах блока (т.е. он является закрытым для данной функции).

В примере используются файлы в текстовом режиме; путем применения режимов "ab+" и "rb" можно было бы обрабатывать двоичные файлы.

Произвольный доступ с двоичным вводом-выводом

Произвольный доступ чаще всего применяется с двоичными файлами, записанными с использованием двоичного ввода-вывода, поэтому давайте рассмотрим короткий пример. Программа в листинге 13.6 создает файл с числами типа double и затем предоставляет доступ к его содержимому.

Листинг 13.6. Программа randbin.c

```
/* randbin.c -- произвольный доступ, двоичный ввод-вывод */
#include <stdio.h>
#include <stdlib.h>
#define ARSIZE 1000

int main()
{
   double numbers[ARSIZE];
   double value;
```

```
    const char * file = "numbers.dat";
    int i;
    long pos;
    FILE *iofile;

    // создание набора значений double
    for(i = 0; i < ARSIZE; i++)
        numbers[i] = 100.0 * i + 1.0 / (i + 1);
    // попытка открыть файл
    if ((iofile = fopen(file, "wb")) == NULL)
    {
        fprintf(stderr, "Не удается открыть файл %s для вывода.\n", file);
        exit(EXIT_FAILURE);
    }
    // запись в файл массива в двоичном формате
    fwrite(numbers, sizeof (double), ARSIZE, iofile);
    fclose(iofile);
    if ((iofile = fopen(file, "rb")) == NULL)
    {
        fprintf(stderr,
              "Не удается открыть файл %s для произвольного доступа.\n", file);
        exit(EXIT_FAILURE);
    }
    // чтение избранных элементов из файла
    printf("Введите индекс в диапазоне 0-%d.\n", ARSIZE - 1);
    while (scanf("%d", &i) == 1 && i >= 0 && i < ARSIZE)
    {
        pos = (long) i * sizeof(double); // вычисление смещения
        fseek(iofile, pos, SEEK_SET);    // переход в нужную позицию
        fread(&value, sizeof (double), 1, iofile);
        printf("По этому индексу находится значение %f.\n", value);
        printf("Введите следующий индекс (или значение за пределами диапазона для
завершения):\n");
    }
    // завершение
    fclose(iofile);
    puts("Программа завершена.");
    return 0;
}
```

Первым делом программа создает массив и помещает в него ряд значений. Затем она создает файл по имени `numbers.dat` в двоичном режиме и применяет функцию `fwrite()` для копирования содержимого массива в этот файл. 64-битовая последовательность для каждого значения `double` копируется из памяти в файл. Вы не сможете прочитать результирующий двоичный файл в текстовом редакторе, т.к. эти значения не транслируются в строки. Однако каждое значение хранится в файле точно так же, как оно хранилось в памяти, поэтому точность не теряется. Более того, каждое значение занимает 64 бита пространства в файле, благодаря чему легко вычислять местонахождение каждого значения.

Во второй части программы файл открывается для чтения и пользователю предлагается ввести индекс значения. Умножение значения индекса на количество байтов, занимаемых типом `double`, дает позицию в файле. Далее в программе вызывается `fseek()` для перехода в эту позицию и `fread()` для чтения значения из этого места. Обратите внимание на отсутствие спецификаторов формата. Взамен `fread()` копирует 8 байтов, начиная с заданной позиции, в ячейку памяти, указанную `&value`. После этого программа использует функцию `printf()` для отображения `value`.

Ниже показаны результаты пробного запуска:

```
Введите индекс в диапазоне 0-999.
500
По этому индексу находится значение 50000.001996.
Введите следующий индекс (или значение за пределами диапазона для
завершения):
900
По этому индексу находится значение 90000.001110.
Введите следующий индекс (или значение за пределами диапазона для
завершения):
0
По этому индексу находится значение 1.000000.
Введите следующий индекс (или значение за пределами диапазона для
завершения):
-1
Программа завершена.
```

Ключевые понятия

Программа С рассматривает ввод как поток байтов; источником этого потока может быть файл, устройство ввода (такое как клавиатура) или даже вывод из другой программы. Подобным же образом программа С трактует вывод как поток байтов; местом назначения может быть файл, экран монитора и т.д.

То, как в С интерпретируется входной или выходной поток байтов, зависит от применяемых функций ввода-вывода. Программа может читать и сохранять байты без их изменений либо интерпретировать байты как символы, которые, в свою очередь, могут быть интерпретированы как обычный текст или текстовое представление чисел. Аналогично, при выводе используемые функции определяют, передаются ли двоичные значения без изменений либо преобразуются в текст или текстовое представление чисел. Если есть числовые данные, которые вы хотите сохранять и затем восстанавливать без потери точности, применяйте двоичный режим и функции `fread()` и `fwrite()`. Если вы сохраняете текстовую информацию и хотите создать файл, который может быть просмотрен с помощью обычных текстовых редакторов, используйте текстовый режим и такие функции, как `getc()` и `fprintf()`.

Для доступа в файл вам потребуется создать указатель файла (типа `FILE *`) и связать его с конкретным именем файла. Для работы с файлом в последующем коде будет применяться этот указатель, а не имя файла.

Важно понимать, как в С поддерживается концепция конца файла. Обычно в программе для чтения файла используется цикл для чтения входных данных до тех пор, пока не будет достигнут конец файла. Функции ввода С не обнаруживают конец файла до тех пор, пока они не предпримут попытку чтения за концом файла. Это означает, что проверка на предмет конца файла должна производиться непосредственно *после* попытки чтения. В качестве руководства можете применять модели ввода из двух файлов, помеченные как "правильное проектное решение" в разделе "Конец файла" данной главы.

Резюме

Запись и чтение из файлов является важной частью большинства программ на С. Многие реализации С предлагают для этих целей службы низкоуровневого и стандартного высокоуровневого ввода-вывода.

Поскольку библиотека ANSI C включает стандартные, но не низкоуровневые службы ввода-вывода, стандартный пакет обладает большей переносимостью.

Стандартный пакет ввода-вывода автоматически создает буферы для ввода и вывода для ускорения передачи данных. Функция fopen() открывает файл для стандартного ввода-вывода и создает структуры данных, предназначенные для хранения информации о файле и буфере. Функция fopen() возвращает указатель на такую структуру данных, и этот указатель используется другими функциями для идентификации файла, подлежащего обработке. Функции feof() и ferror() сообщают о причине отказа операции ввода-вывода.

Ввод в C рассматривается как поток байтов. Если вы применяете функцию fread(), то ввод трактуется как двоичные значения, которые должны быть помещены в указанное место памяти. Если вы используете fscanf(), getc(), fgets() или любые родственные им функции, то каждый байт рассматривается как код символа. Функции fscanf() и scanf() затем пытаются преобразовать этот код символа в другие типы, как отражено спецификаторами формата.

Например, входное значение 23 спецификатор %f преобразует в значение с плавающей запятой, спецификатор %d – в целочисленное значение, а спецификатор %s – в строку. Семейство функций getc() и fgets() оставляет ввод в виде кодов символов и сохраняет его либо в переменных типа char как отдельные символы, либо в массивах char как строки. Подобным же образом функция fwrite() помещает двоичные данные непосредственно в выходной поток, тогда как другие функции вывода перед помещением в поток вывода преобразуют несимвольные данные в символьные представления.

Стандарт ANSI C предоставляет два режима открытия файла: двоичный и текстовый. Когда файл открыт в двоичном режиме, его можно читать байт за байтом. Когда файл открыт в текстовом режиме, его содержимое может быть отображено из системного представления текста в представление C. Для систем Unix и Linux эти два режима идентичны.

Функции ввода getc(), fgets(), fscanf() и fread() обычно читают файл последовательно, стартуя с начала файла. Тем не менее, функции fseek() и ftell() позволяют программе перемещаться в любую позицию внутри файла, делая возможным произвольный доступ. Функции fgetpos() и fsetpos() распространяют аналогичную возможность на файлы больших размеров. В двоичном режиме произвольный доступ работает лучше, чем в текстовом режиме.

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля приведены в приложении А.

1. Что неправильно в следующей программе:

```
int main(void)
{
   int * fp;
   int k;

   fp = fopen("gelatin");
   for (k = 0; k < 30; k++)
      fputs(fp, "Кто-то ест студень.");
   fclose("gelatin");
   return 0;
}
```

2. Что делает следующая программа? (Предположите, что она запускается в среде командной строки.)

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
int main(int argc, char *argv[])
{
    int ch;
    FILE *fp;

    if (argc < 2)
      exit(EXIT_FAILURE);
    if ( (fp = fopen(argv[1], "r")) == NULL)
        exit(EXIT_FAILURE);
    while ( (ch= getc(fp)) != EOF )
       if( isdigit(ch) )
          putchar(ch);
    fclose (fp);

    return 0;
}
```

3. Предположим, что в программе имеются следующие операторы:

```
#include <stdio.h>
FILE * fp1,* fp2;
char ch;

fp1 = fopen("terky", "r");
fp2 = fopen("jerky", "w");
```

Кроме того, предположим, что оба файла были успешно открыты. Добавьте недостающие аргументы в следующие вызовы функций:

а. `ch = getc();`

б. `fprintf( ,"%c\n", );`

в. `putc( , );`

г. `fclose(); /* закрыть файл terky */`

4. Напишите программу, которая принимает ноль или один аргумент командной строки. Если имеется один аргумент, он интерпретируется как имя файла. Если аргумент не указан, то должен использоваться стандартный ввод (`stdin`). Предположите, что ввод состоит целиком из чисел с плавающей запятой. Программа должна вычислять и отображать среднее арифметическое значение для введенных чисел.

5. Напишите программу, которая принимает два аргумента командной строки. Первым аргументом является символ, а вторым — имя файла. Программа должна выводить из файла только те строки, которые содержат указанный символ.

На заметку!

Строки файла идентифицируются символом новой строки `'\n'`. Предположим, что ни одна из строк по длине не превышает 256 символов. Возможно, потребуется использовать функцию `fgets()`.

6. В чем разница между двоичными и текстовыми файлами с одной стороны и двоичными и текстовыми потоками — с другой?

7. а. В чем разница между сохранением числа 8238201 с помощью fprintf() и его сохранением посредством fwrite()?

   б. В чем разница между сохранением символа S с помощью putc() и его сохранением посредством fwrite()?

8. Чем отличаются друг от друга следующие операторы?

```
printf("Здравствуйте, %s\n", name);
fprintf(stdout, "Здравствуйте, %s\n", name);
fprintf(stderr, "Здравствуйте, %s\n", name);
```

9. Режимы "a+", "r+" и "w+" открывают файл для чтения и записи. Какой из них лучше всего подходит для изменения содержимого, находящегося в файле?

Упражнения по программированию

1. Модифицируйте программу в листинге 13.1, чтобы она предлагала пользователю ввести имя файла и читала его ответ вместе использования аргументов командной строки.

2. Напишите программу копирования файлов, которая получает имя исходного файла и имя копии из командной строки. Используйте стандартный ввод-вывод и двоичный режим, если это возможно.

3. Напишите программу копирования файлов, которая предлагает пользователю ввести имя текстового файла, выступающего в роли исходного, и имя выходного файла. Программа должна использовать функцию toupper() из ctype.h для перевода текста в верхний регистр во время его записи в выходной файл. Применяйте стандартный ввод-вывод и текстовый режим.

4. Напишите программу, которая последовательно отображает на экране содержимое всех файлов, перечисленных в командной строке. Для управления циклом используйте argc.

5. Модифицируйте программу в листинге 13.6, чтобы вместо интерактивного интерфейса она использовала интерфейс командной строки.

6. Программы, работающие с аргументами командной строки, полагаются на то, что пользователь помнит, как их правильно запускать. Перепишите программу из листинга 13.2 так, чтобы вместо использования аргументов командной строки она предлагала пользователю ввести необходимую информацию.

7. Напишите программу, которая открывает два файла. Получать имена файлов можно либо через командную строку, либо предложив пользователю ввести их.

   а. Сделайте так, чтобы эта программа выводила строку 1 первого файла, строку 1 второго файла, строку 2 первого файла, строку 2 второго файла и т.д., пока не будет выведена последняя строка более длинного (по количеству строк) файла.

   б. Модифицируйте программу так, чтобы строки с одним и тем же номером выводились в одной экранной строке.

8. Напишите программу, которая принимает в качестве аргументов командной строки символ и ноль или более имен файлов. Если за символом не следуют аргументы, программа должна читать стандартный ввод. В противном случае она должна открывать каждый файл по очереди и сообщать, сколько раз в нем встре-

чается указанный символ. Вместе с результатом каждого подсчета должны быть указаны имя файла и сам символ. Предусмотрите средства проверки ошибок, позволяющие выяснить корректность количества аргументов и возможность открытия файлов. Если файл не может быть открыт, программа должна сообщать об этом факте и переходить к обработке следующего файла.

9. Модифицируйте программу в листинге 13.3, чтобы каждое слово было пронумеровано в соответствии с порядком его добавления в список, начиная с 1. Позаботьтесь о том, чтобы при втором запуске программы новая нумерация слов начиналась с того места, где была закончена нумерация при предыдущем запуске.

10. Напишите программу, открывающую текстовый файл, имя которого получается интерактивно. Организуйте цикл, который предлагает пользователю ввести позицию в файле. Затем программа должна вывести часть файла, начиная с этой позиции и заканчивая следующим символом новой строки. Цикл ввода должен завершаться при вводе отрицательного или нечислового значения.

11. Напишите программу, которая принимает два аргумента командной строки. Первый аргумент — это строка, а второй — имя файла. Программа должна искать в файле и выводить все строки, содержащие указанную. Поскольку задача ориентирована на строки, а не на символы, используйте функцию `fgets()` вместо `getc()`. Для поиска строки, содержащей заданную, применяйте функцию `strstr()` (кратко описанную в упражнении 7 из главы 11) из стандартной библиотеки C.

12. Создайте текстовый файл, состоящий из 20 строк по 30 целых чисел. Целые значения должны входить в диапазон от 0 до 9 и отделяться друг от друга пробелами. Файл является цифровым представлением изображения, при этом значения от 0 до 9 воспроизводят возрастающие уровни темноты. Напишите программу, которая читает содержимое этого файла в массив элементов `int` с размером 20 на 30. При грубом подходе к преобразованию такого цифрового представления в изображение программа должна использовать значения из этого массива для инициализации массива элементов `char` с размером 20 на 31. При этом 0 соответствует символу пробела, 1 — символу точки и т.д.; большее значение представляется символом, занимающим большую площадь. Например, символ # может использоваться для представления значения 9. Последний символ (31-й) в каждой строке должен быть нулевым, образуя в итоге массив из 20 строк. Программа должна отобразить результирующее изображение (т.е. вывести строки) и также сохранить результат в текстовом файле. Например, предположим, что начальные данные выглядят так:

```
0 0 9 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 9 0 0 0 0 0 0 0 5 8 9 9 8 5 5 2 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 1 9 8 5 4 5 2 0 0 0 0 0 0 0 0 0
0 0 0 0 9 0 0 0 0 0 0 0 5 8 9 9 8 5 0 4 5 2 0 0 0 0 0 0 0 0
0 0 9 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 4 5 2 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 1 8 5 0 0 0 4 5 2 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 4 5 2 0 0 0 0 0
5 5 5 5 5 5 5 5 5 5 5 5 5 8 9 9 8 5 5 5 5 5 5 5 5 5 5 5 5 5
8 8 8 8 8 8 8 8 8 8 8 8 5 8 9 9 8 5 8 8 8 8 8 8 8 8 8 8 8 8
9 9 9 9 0 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 3 9 9 9 9 9 9 9 9
8 8 8 8 8 8 8 8 8 8 8 8 5 8 9 9 8 5 8 8 8 8 8 8 8 8 8 8 8 8
```

```
5 5 5 5 5 5 5 5 5 5 5 5 5 8 9 9 8 5 5 5 5 5 5 5 5 5 5 5 5 5
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 6 6 0 0 0 0 0 0
0 0 0 0 2 2 0 0 0 0 0 0 5 8 9 9 8 5 0 0 5 6 0 0 6 5 0 0 0 0
0 0 0 0 3 3 0 0 0 0 0 0 5 8 9 9 8 5 0 5 6 1 1 1 1 6 5 0 0 0
0 0 0 0 4 4 0 0 0 0 0 0 5 8 9 9 8 5 0 0 5 6 0 0 6 5 0 0 0 0
0 0 0 0 5 5 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 6 6 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 0 0 0 0 0 0 0 0
```

При заданном выборе выходных символов вывод будет иметь следующий вид:

```
  #      *%##%*'
   #     *%##%**'
         *%.#%*~*'
   #     *%##%* ~*'
  #      *%##%*  ~*'
         *%#.%*   ~*'
         *%##%*    ~*'
*************%##%*************
%%%%%%%%%%%%*%##%*%%%%%%%%%%%%
#### #################:#######
%%%%%%%%%%%%*%##%*%%%%%%%%%%%%
*************%##%*************
         *%##%*
         *%##%*    ==
   ''    *%##%*  *=  =*
   ::    *%##%* *=....=*
   ~~    *%##%*  *=  =*
   **    *%##%*    ==
         *%##%*
         *%##%*
```

13. Выполните упражнение 12, используя массивы переменной длины вместо стандартных массивов.

14. Цифровые изображения, особенно передаваемые по радиоканалу космическим аппаратом, могут содержать помехи. Добавьте в программу из упражнения 12 функцию подавления помех. Она должна сравнивать каждое значение с соседними значениями, расположенными слева и справа, снизу и сверху. Если значение отличается более чем на 1 от своих соседей, оно должно быть заменено средним значением всех соседних величин. Среднее значение должно быть округлено до ближайшего целого. Обратите внимание, что точки, находящиеся на границах, имеют меньше четырех соседей, поэтому они требуют специальной обработки.

14

Структуры и другие формы данных

В ЭТОЙ ГЛАВЕ...

- Ключевые слова: struct, union, typedef
- Операции: . ->
- Структуры в языке С и способы создания шаблонов и переменных типа структур
- Доступ к членам структуры и написание функций для обработки структур
- Средство typedef в языке С
- Объединения и указатели на функции

Одним из наиболее важных шагов при проектировании программы является выбор подходящего способа представления данных. Во многих случаях простой переменной или даже массива оказывается недостаточно. Язык С позволяет расширить возможности представления данных с помощью *переменных типа структур*. В своей базовой форме структура С является достаточно гибким средством, чтобы представлять широкое разнообразие данных, и она позволяет изобретать новые формы. Если вы знакомы с записями в языке Pascal, то вам будет легко освоиться со структурами. Если же нет, то настоящая глава послужит введением в структуры С. Давайте рассмотрим конкретный пример, который покажет, почему могут понадобиться структуры, и продемонстрирует их создание и применение.

Учебная задача: создание каталога книг

Гвен Глен желает сформировать каталог своих книг. Она хотела бы располагать разнообразной информацией по каждой книге: название, автор, издательство, дата регистрации авторского права, количество страниц и стоимость книги. Некоторые из этих элементов данных, такие как названия, могут храниться в массивах строк. Другие элементы требуют массива значений типа `int` или `float`. При наличии семи разных массивов отслеживание всех данных может стать затруднительным, особенно если учесть, что Гвен заинтересована в генерации нескольких списков книг: с сортировкой по названию, по авторам, по цене и т.д. Самое лучшее решение предусматривает использование одного массива, каждый элемент которого содержит полные сведения об одной книге.

Затем Гвен понадобится форма данных, которая может содержать строки и числа, но каким-то образом разделяя эту информацию. Структура С отвечает таким требованиям. Чтобы посмотреть, как создать структуру подобного рода, и каким образом она работает, мы начнем с ограниченного примера. Для упрощения задачи мы наложим два ограничения. Во-первых, мы будем включать только название, автора и текущую стоимость. Во-вторых, мы ограничим каталог одной книгой. Однако не стоит переживать по поводу этого ограничения, поскольку вскоре мы расширим программу.

Взгляните на программу, показанную в листинге 14.1, и на ее вывод. Затем прочитайте объяснение основных ее особенностей.

Листинг 14.1. Программа `book.c`

```
/* book.c -- каталог для одной книги */
#include <stdio.h>
#include <string.h>
char * s_gets(char * st, int n);
#define MAXTITL 41      /* максимальная длина названия + 1 */
#define MAXAUTL 31      /* максимальная длина имени автора + 1 */

struct book {           /* шаблон структуры: дескриптором является book */
   char title[MAXTITL];
   char author[MAXAUTL];
   float value;
};                      /* конец шаблона структуры */

int main(void)
{
   struct book library; /* объявление library в качестве переменной типа book */

   printf("Введите название книги.\\n");
   s_gets(library.title, MAXTITL); /* доступ к разделу названия книги */
```

```
    printf("Теперь введите ФИО автора.\n");
    s_gets(library.author, MAXAUTL);
    printf("Теперь введите цену.\n");
    scanf("%f", &library.value);
    printf("%s авторства %s: $%.2f\n", library[index].title,
        library.author, library.value);
    printf("%s: \"%s\" ($%.2f)\n", library.author,
        library.title, library.value);
    printf("Готово.\n");
    return 0;
}
char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');      // поиск новой строки
        if (find)                     // если адрес не равен NULL,
            *find = '\0';             // поместить туда нулевой символ
        else
            while (getchar() != '\n')
                continue;             // отбросить остаток строки
    }
    return ret_val;
}
```

Как и в предыдущих главах, мы применяем функцию `s_gets()` для удаления символа новой строки, который функция `fgets()` обычно оставляет во входной строке. Ниже приведены результаты пробного запуска:

```
Введите название книги.
Chicken of the Andes
Теперь введите ФИО автора.
Disma Lapoult
Теперь введите цену.
29.99
Chicken of the Andes авторства Disma Lapoult: $29.99
Disma Lapoult: "Chicken of the Andes" ($29.99)
Готово.
```

Структура, созданная в листинге 14.1, имеет три части (называемые членами или полями) – для хранения названия книги, для хранения имени автора и для хранения цены. Вы должны овладеть следующими тремя навыками:

- настройка формата или схемы для структуры;
- объявление переменной, соответствующей такой схеме;
- обеспечение доступа к индивидуальным компонентам переменной типа структуры.

Объявление структуры

Объявление структуры представляет собой генеральный план, который описывает способ формирования структуры. Объявление структуры выглядит следующим образом:

```
struct book {
   char title[MAXTITL];
   char author[MAXAUTL];
   float value;
};
```

Это объявление описывает структуру, образованную из двух символьных массивов и одной переменной типа float. Оно не создает реальный объект данных, но определяет, из чего состоит такой объект. (Иногда мы будем ссылаться на объявление структуры как на *шаблон*, потому что оно показывает, каким образом будут храниться данные. Если вы слышали о шаблонах C++, то знайте, что они представляют собой другой, более претенциозный случай использования этого понятия.) Давайте обратимся к деталям объявления. Первым идет ключевое слово struct. Оно указывает, что за ним следует структура. Далее находится необязательный *дескриптор* — слово book — сокращенная метка, которую можно применять для ссылки на эту структуру. Таким образом, позже мы имеем следующее объявление:

```
struct book library;
```

Оно объявляет library как переменную типа структуры, которая использует схему структуры book.

После этого в объявлении структуры указан список членов, заключенный в фигурные скобки. Каждый член описан собственным объявлением, которое оканчивается точкой с запятой. Например, порция названия книги (title) представляет собой массив, содержащий MAXTITL элементов типа char. Членом структуры может быть любой тип данных С, в том числе и другая структура.

Точка с запятой после закрывающей фигурной скобки завершает определение шаблона структуры. Это объявление можно разместить за пределами любой функции (внешне), как было сделано в здесь, либо внутри определения функции. Если объявление структуры находится внутри функции, ее дескриптор может применяться только в рамках этой функции. Если объявление является внешним, оно доступно всем функциям, которые следуют за этим объявлением в файле. Например, во второй функции можно было бы определить:

```
struct book dickens;
```

и в этой функции появилась бы переменная dickens, имеющая ту же самую форму, что и структура book.

Имя дескриптора указывать не обязательно, но оно должно использоваться, когда шаблон структуры определяется в одном месте, а фактические переменные — в других местах. Мы возвратимся к этому вопросу позже, после того как посмотрим на определение переменных типа структур.

Определение переменной типа структуры

Понятие *структура* применяется в двух смыслах. Одним из них является "схема структуры" — то, что мы недавно обсуждали. Схема структуры сообщает компилятору, *как* представлять данные, но она не приводит к *выделению* пространства в памяти для этих данных. Следующий шаг заключается в создании *переменной типа структуры*, и в этом состоит второй смысл понятия. Строка программы, создающая переменную типа структуры, имеет вид:

```
struct book library;
```

Обнаружив эту инструкцию, компилятор создает переменную library. Используя шаблон book, компилятор выделяет память для массива из MAXTITL элементов типа char, для массива из MAXAUTL элементов типа char и для переменной float. Эта память объединена в единую конструкцию под общим именем library, как показано на рис. 14.1. (В следующем разделе объясняется, каким образом при необходимости разделять эту конструкцию.)

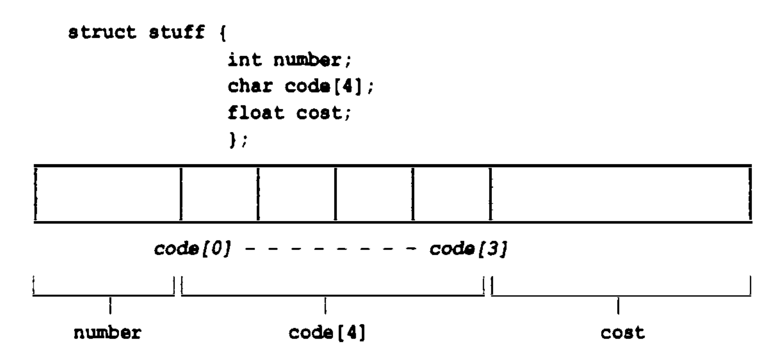


Рис. 14.1. Выделение памяти под структуру

В объявлении переменной типа структуры конструкция struct book играет ту же самую роль, что и ключевые слова int или float в более простых объявлениях. Например, можно было бы объявить две переменные типа struct book и даже указатель на структуру такого вида:

```
struct book doyle, panshin, * ptbook;
```

Каждая переменная doyle и panshin типа структуры будет иметь три части: title, author и value. Указатель ptbook может указывать на переменные doyle, panshin или на любую другую структуру book. По существу объявление структуры book создает новый тип по имени struct book.

С точки зрения компьютера объявление

```
struct book library;
```

является сокращением для

```
struct book {
   char title[MAXTITL];
   char author[AXAUTL];
   float value;
} library;      /* объявление с именем переменной */
```

Другими словами, процесс объявления структуры и процесс определения переменной типа структуры можно объединить в одно действие. Комбинация объявления и определений переменных делает излишним применение дескриптора:

```
struct {        /* дескриптор отсутствует */
   char title[MAXTITL];
   char author[MAXAUTL];
   float value;
} library;
```

Тем не менее, дескриптор понадобится, если шаблон структуры должен использоваться более одного раза. Можно также применить альтернативный вариант с typedef, рассматриваемый далее в этой главе.

Есть еще один аспект, касающийся определения переменной типа структуры, который не был задействован в этом примере: инициализация. Давайте взглянем на нее.

Инициализация структуры

Вы уже видели, как инициализировать переменные и массивы:

```
int count = 0;
int fibo[7] = {0,1,1,2,3,5,8};
```

Можно ли также инициализировать переменную типа структуры? Да, можно. Для инициализации структуры (любого класса хранения ANSI C и последующих стандартов, но исключая автоматические переменные в реализациях, которые предшествуют ANSI C) используется синтаксис, аналогичный применяемому при инициализации массивов:

```
struct book library = {
    "The Pious Pirate and the Devious Damsel",
    "Renee Vivotte",
    1.95
};
```

Как видите, используется список разделенных запятыми инициализаторов, заключенный в фигурные скобки. Тип каждого инициализатора должен соответствовать типу члена структуры, который он инициализирует. Следовательно, член `title` можно инициализировать строкой, а член `value` — числом. Чтобы сделать эти связи более очевидными, мы выделили для каждого члена отдельную строку инициализации, но компилятору вполне достаточно отделения инициализаторов друг от друга запятыми.

НА ЗАМЕТКУ! Инициализация структуры и продолжительность хранения

В главе 12 упоминалось, что при инициализации переменной со статической продолжительностью хранения (например, статической с внешним связыванием, статической с внутренним связыванием или статической без связывания) должны использоваться константные значения. Это применимо также к структурам. Если вы инициализируете структуру со статической продолжительностью хранения, то значения в списке инициализаторов должны быть константными выражениями. Если продолжительность хранения является автоматической, значения в списке инициализаторов не обязательно должны быть константами.

Доступ к членам структуры

Структура похожа на "супермассив", в котором один элемент может иметь тип `char`, другой — `float`, а следующий — массив элементов типа `int`. Обращаться к отдельным элементам массива можно с помощью индекса. А как получить доступ к индивидуальным членам структуры? Для этого служит операция членства в структуре — точка (`.`). Например, `library.value` — это компонент `value` структуры `library`. Конструкцию `library.value` можно использовать подобно любой другой переменной типа `float`. Аналогично, `library.title` можно применять в точности как массив типа `char`. По этой причине в приведенной выше программе используются такие выражения, как

```
s_gets(library.title, MAXTITL);
```

и

```
scanf("%f", &library.value);
```

По существу `.title`, `.author` и `.value` играют роль индексов для структуры `book`.

Обратите внимание, что хотя library – это структура, library.value имеет тип float и применяется подобно любой другой переменной типа float. Например, scanf("%f",...) требует адреса ячейки со значением float, и именно таким адресом является &library.float. Здесь операция точки имеет более высокий приоритет, чем операция &, поэтому выражение эквивалентно &(library.float).

При наличии второй переменной структуры того же типа можно воспользоваться тем же самым методом:

```
struct book bill, newt;

s_gets(bill.title, MAXTITL);
s_gets(newt.title, MAXTITL);
```

Конструкция .title относится к первому члену структуры book. Следует отметить, что исходная программа выводит содержимое структуры library в двух разных форматах. Это иллюстрирует свободу, доступную при работе с членами структуры.

Инициализаторы для структур

Стандарты С99 и С11 предоставляют назначенные инициализаторы для структур. Синтаксис похож на синтаксис назначенных инициализаторов для массивов. Однако назначенные инициализаторы для структур при идентификации конкретных членов используют операцию точки и имена членов, а не квадратные скобки и индексы. Например, чтобы инициализировать только член value структуры book, можно поступить так:

```
struct book surprise = {.value = 10.99};
```

Назначенные инициализаторы можно указывать в любом порядке:

```
struct book gift = {.value = 25.99,
                    .author = "James Broadfool",
                    .title = "Rue for the Toad"};
```

Как и в случае массивов, обычный инициализатор, следующий за назначенным, присваивает значение члену, который следует за членом, указанным в назначенном инициализаторе. Кроме того, член получает значение, которое было предоставлено последним. Например, взгляните на такое объявление:

```
struct book gift= {.value = 18.90,
                   .author = "Philionna Pestle",
                   0.25};
```

Значение 0.25 присваивается члену value, поскольку он находится непосредственно после члена author в объявлении структуры. Новое значение 0.25 заменяет собой указанное ранее значение 18.90. Теперь, располагая базовыми знаниями, вы готовы расширить свой кругозор и ознакомиться с несколькими типами, в которых задействованы структуры. Вы увидите массивы структур, структуры структур, указатели на структуры и функции, которые обрабатывают структуры.

Массивы структур

Давайте расширим программу каталога книг для поддержки большего количества книг. Очевидно, что каждая книга может быть описана одной переменной типа book. Чтобы описать две книги, необходимы две такие переменные и т.д. Для поддержки нескольких книг понадобится массив структур подобного рода, и его мы создадим в программе, показанной в листинге 14.2. (Если вы имеете дело с Borland C/C++, ознакомьтесь с врезкой "Borland C и плавающая запятая" далее в главе.)

Структуры и память

В программе manybook.c применяется массив из 100 структур. Поскольку массив является объектом с автоматическим классом хранения, информация обычно размещается в стеке. Крупный массив такого рода требует области памяти приличного размера, что может вызвать проблемы. Если во время выполнения вы получаете сообщение об ошибке, возможно, уведомляющее о переполнении стека, то ваш компилятор, скорее всего, использует стандартный размер для стека, который слишком мал для этого примера. Чтобы исправить положение, вы можете с помощью опций компилятора установить размер стека в 10000, обеспечив достаточное место для данного массива структур, или же сделать массив статическим либо внешним (тогда он не будет размещаться в стеке); можно также уменьшить размер массива, к примеру, до 16. А почему мы изначально не выбрали массив небольшого размера? Причина в том, что вы должны знать об этой потенциальной проблеме с размером стека и уметь справляться с ней, когда она возникнет в будущей практике.

Листинг 14.2. Программа manybook.c

```
/* manybook.c -- каталог для нескольких книг */
#include <stdio.h>
#include <string.h>
char * s_gets(char * st, int n);
#define MAXTITL 40
#define MAXAUTL 40
#define MAXBKS 100              /* максимальное количество книг */

struct book {                   /* установка шаблона book      */
   char title[MAXTITL];
   char author[MAXAUTL];
   float value;
};

int main(void)
{
   struct book library[MAXBKS]; /* массив структур типа book    */
   int count = 0;
   int index;

   printf("Введите название книги.\n");
   printf("Нажмите [enter] в начале строки, чтобы закончить ввод.\n");
   while (count < MAXBKS && s_gets(library[count].title, MAXTITL) != NULL
        && library[count].title[0] != '\0')
   {
      printf("Теперь введите ФИО автора.\n");
      s_gets(library[count].author, MAXAUTL);
      printf("Теперь введите цену.\n");
      scanf("%f", &library[count++].value);
      while (getchar() != '\n')
         continue;          /* очистить входную строку */
      if (count < MAXBKS)
         printf("Введите название следующей книги.\n");
   }

   if (count > 0)
   {
      printf("Каталог ваших книг:\n");
      for (index = 0; index < count; index++)
         printf("%s авторства %s: $%.2f\n", library[index].title,
            library[index].author, library[index].value);
   }
```

```
   else
      printf("Вообще нет книг? Очень плохо.\n");

   return 0;
}

char * s_gets(char * st, int n)
{
   char * ret_val;
   char * find;

   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      find = strchr(st, '\n');     // поиск новой строки
      if (find)                    // если адрес не равен NULL,
         *find = '\0';             // поместить туда нулевой символ
      else
         while (getchar() != '\n')
            continue;              // отбросить остаток строки
   }
   return ret_val;
}
```

Borland C и плавающая запятая

Ранние компиляторы Borland C пытались делать программы более компактными за счет применения сокращенной версии функции `scanf()`, если в программах не использовались значения с плавающей запятой. Тем не менее, эти компиляторы (до версии Borland C/C++ 3.1 для DOS, но не Borland C/C++ 4.0) допускают просчет, если значения с плавающей запятой содержатся только внутри массива структур, как в случае листинга 14.2. В результате вы получаете сообщение, подобное следующему:

```
scanf : floating point formats not linked
Abnormal program termination
scanf : форматы с плавающей запятой не подключены
Аварийное завершение программы
```

Это можно обойти путем добавления в программу такого кода:

```
#include <math.h>
double dummy = sin(0.0);
```

Данный код приводит к принудительной загрузке версии `scanf()` с плавающей запятой.

Вот результаты пробного запуска:

```
Введите название книги.
Нажмите [enter] в начале строки, чтобы закончить ввод.
My Life as a Budgie
Теперь введите ФИО автора.
Mack Zackles
Теперь введите цену.
12.95
Введите название следующей книги.
...ввод информации о других книгах...
Каталог ваших книг:
My Life as a Budgie авторства Mack Zackles: $12.95
Thought and Unthought Rethought авторства Kindra Schlagmeyer: $43.50
Concerto for Financial Instruments авторства Filmore Walletz: $49.99
The CEO Power Diet авторства Buster Downsize: $19.25
```

```
C++ Primer Plus авторства Stephen Prata: $59.99
Fact Avoidance: Perception as Reality авторства Polly Bull: $19.97
Coping with Coping авторства Dr. Rubin Thonkwacker: $0.02
Diaphanous Frivolity авторства Neda McFey: $29.99
Murder Wore a Bikini авторства Mickey Splats: $18.95
A History of Buvania, Volume 8 авторства Prince Nikoli Buvan: $50.04
Mastering Your Digital Watch, 5nd Edition авторства Miklos Mysz: $28.95
A Foregone Confusion авторства Phalty Reasoner: $5.99
Outsourcing Government: Selection vs. Election авторства Ima Pundit: $33.33
```

Для начала мы объясним, как объявлять массивы структур и получать доступ к индивидуальным членам. После этого мы проанализируем два аспекта программы.

Объявление массива структур

Объявление массива структур подобно объявлению любого другого вида массива, например:

```
struct book library[MAXBKS];
```

Здесь `library` объявляется как массив, содержащий `MAXBKS` элементов. Каждый элемент этого массива является структурой типа `book`. Таким образом, `library[0]` — одна структура типа `book`, `library[1]` — вторая структура типа `book` и т.д.

На рис. 14.2 приведена иллюстрация, которая способствует лучшему пониманию сказанного. Само имя `library` не является именем структуры; оно представляет собой имя массива, элементы которого — структуры типа `struct book`.

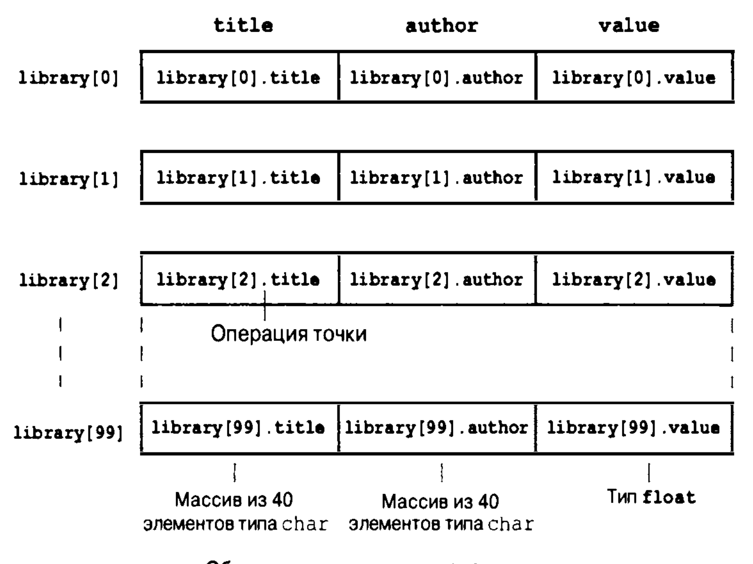


Рис. 14.2. Массив структур

Идентификация членов в массиве структур

Для идентификации членов в массиве структур применяются те же самые правила, что и в случае индивидуальных структур: за именем структуры должна следовать операция точки, а затем имя члена.

Например:

```
library[0].value /*значение value, ассоциированное с первым элементом массива*/
library[4].title /*значение title, ассоциированное с пятым элементом массива*/
```

Обратите внимание, что индекс массива указан после `library`, а не после имени члена:

```
library.value[2]    // НЕПРАВИЛЬНО
library[2].value    // ПРАВИЛЬНО
```

Причина использования конструкции `library[2].value` состоит в том, что `library[2]` является именем переменной типа структуры, точно так же, как `library[1]` – это имя еще одной переменной типа структуры.

Кстати, что представляет следующая конструкция?

```
library[2].title[4]
```

Это пятый символ в названии книги (часть `title[4]`), которую описывает третья структура (часть `library[2]`). В приведенном примере это будет символ *B*. Данный пример показывает, что индексы, находящиеся справа от операции точки, применяются к индивидуальным членам, но индексы, расположенные слева от операции точки, относятся к массиву структур.

В итоге мы допустимы такие операторы:

```
library             // массив структур типа book
library[2]          // элемент массива, т.е. структура book
library[2].title    // символьный массив (член title элемента library[2])
library[2].title[4] // символ в массиве члена title
```

Давайте завершим рассмотрение программы.

Анализ программы

Основное отличие этой программы от первой заключается в том, что в ней включен цикл для чтения множества записей. Цикл начинается со следующего условия `while`:

```
while (count < MAXBKS && s_gets(library[count].title, MAXTITL) != NULL
                      && library[count].title[0] != '\0')
```

Выражение `s_gets(library[count].title, MAXTITL)` читает строку для названия книги; это выражение принимает значение `NULL`, если функция `s_gets()` пытается прочитать символ, следующий за концом файла. Выражение `library[count].title[0] != '\0'` проверяет, не является ли первый символ строки нулевым (т.е. не пустая ли строка). Если пользователь нажимает клавишу <Enter> в начале строки, передается пустая строка и цикл завершается. В программе также предусмотрена проверка, которая не позволяет вводить больше записей для книг, чем допускает размер массива.

Далее в программе идут следующие строки:

```
while (getchar() != '\n')
   continue;          /* очистить входную строку */
```

Как уже говорилось в предшествующих главах, этот код компенсирует игнорирование функцией `scanf()` пробелов и символов новой строки. Когда вы отвечаете на запрос цены книги, вы набираете что-нибудь такое:

```
12.50[enter]
```

Это приводит к передаче следующей последовательности символов:

12.50\n

Функция scanf() получает символы 1, 2, ., 5 и 0, но оставляет символ \n в ожидании, что им займется следующий оператор чтения. Если бы код очистки входной строки отсутствовал, то следующий оператор чтения, s_gets(library[count].title, MAXTITL), прочитал бы оставленный символ новой строки и посчитал бы, что введена пустая строка, а это является сигналом к прекращению ввода. Вставленный нами код читает символы до тех пор, пока не обнаружит символ новой строки и затем избавится от него. Он ничего не делает с этими символами, а лишь удаляет их из входной очереди. Это позволяет функции s_gets() корректно начать чтение следующих входных данных.

Теперь возвратимся к исследованию структур.

Вложенные структуры

Иногда удобно, чтобы одна структура содержала другую структуру, которая называется *вложенной*. Например, Шейла Пироски создает структуру с информацией о своих друзьях. Одним из членов структуры является, естественно, имя друга. Однако имя само может быть представлено с помощью структуры, с отдельными записями для имени и фамилии. В листинге 14.3 показан сжатый пример работы Шейлы.

Листинг 14.3. Программа friend.c

```
// friend.c -- пример вложенной структуры
#include <stdio.h>
#define LEN 20
const char * msgs[5] =
{
    "   Благодарю вас за чудесно проведенный вечер, ",
    "Вы однозначно продемонстрировали, что ",
    "являет собою особый тип мужчины. Мы обязательно должны встретиться",
    "за восхитительным ужином с ",
    " и весело провести время."
};

struct names {                    // первая структура
    char first[LEN];
    char last[LEN];
};

struct guy {                      // вторая структура
    struct names handle;          // вложенная структура
    char favfood[LEN];
    char job[LEN];
    float income;
};

int main(void)
{
    struct guy fellow = {         // инициализация переменной
        { "Билли", "Бонс" },
        "запеченными омарами",
        "персональный тренер",
        68112.00
    };
```

```
    printf("Дорогой %s, \n\n", fellow.handle.first);
    printf("%s%s.\n", msgs[0], fellow.handle.first);
    printf("%s%s\n", msgs[1], fellow.job);
    printf("%s\n", msgs[2]);
    printf("%s%s%s", msgs[3], fellow.favfood, msgs[4]);
    if (fellow.income > 150000.0)
       puts("!!");
    else if (fellow.income > 75000.0)
       puts("!");
    else
       puts(".");
    printf("\n%40s%s\n ", " ", "До скорой встречи,");
    printf("%40s%s\n", " ", "Шейла");

    return 0;
}
```

Ниже представлен вывод, полученный из этой программы:

```
Дорогой Билли,

   Благодарю вас за чудесно проведенный вечер, Билли.
Вы однозначно продемонстрировали, что персональный тренер
являет собою особый тип мужчины. Мы обязательно должны встретиться
за восхитительным ужином с запеченными омарами и весело провести время.
                                        До скорой встречи,
                                        Шейла
```

Во-первых, обратите внимание на то, каким образом вложенная структура устанавливается в объявлении структуры. Она просто объявляется, как если бы это была переменная типа `int`:

```
struct names handle;
```

Такое объявление говорит о том, что `handle` является переменной типа `struct names`. Разумеется, файл должен также включать объявление структуры `names`.

Во-вторых, посмотрите, как получать доступ к члену вложенной структуры; нужно всего лишь два раза воспользоваться операцией точки:

```
printf("Дорогой %s!\n\n", fellow.handle.first);
```

Если рассматривать эту конструкцию слева направо, она интерпретируется следующим образом:

```
(fellow.handle).first
```

То есть необходимо найти структуру `fellow`, затем член `handle` структуры `fellow` и, наконец, член `first` структуры типа `names`.

Указатели на структуры

Любители указателей будут рады узнать о возможности иметь указатели на структуры. Существуют, по меньшей мере, четыре причины, обусловливающие необходимость в указателях на структуры. Во-первых, точно так же, как манипулировать указателями на массивы проще, чем самими массивами (скажем, в задаче сортировки), с указателями на структуры часто работать легче, чем с самими структурами. Во-вторых, в некоторых старых реализациях структура не может быть передана как аргумент функции, но указатель на структуру — может. В-третьих, несмотря на то, что структуру можно передавать в качестве аргумента, передача указателя зачастую эффективнее.

И, в-четвертых, многие замысловатые представления данных применяют структуры, содержащие указатели на другие структуры.

Следующий краткий пример (листинг 14.4) демонстрирует определение указателя на структуру и его использование для доступа к членам структуры.

Листинг 14.4. Программа `friends.c`

```
/* friends.c -- использование указателя на структуру */
#include <stdio.h>
#define LEN 20

struct names {
   char first[LEN];
   char last[LEN];
};

struct guy {
   struct names handle;
   char favfood[LEN];
   char job[LEN];
   float income;
};

int main(void)
{
   struct guy fellow[2] = {
      {{ "Билли", "Бонс" },
         "запеченными омарами",
         "персональный тренер",
         68112.00
      },
      {{ "Джим", "Хокинс" },
         "рыбным фрикасе",
         "редактор таблоида",
         232400.00
      }
   };
   struct guy * him;   /* указатель на структуру */

   printf("адрес #1: %p #2: %p\n", &fellow[0], &fellow[1]);
   him = &fellow[0];   /* сообщает указателю, на что указывать */
   printf("указатель #1: %p #2: %p\n", him, him + 1);
   printf("him->income равно $%.2f: (*him).income равно $%.2f\n",
       him->income, (*him).income);
   him++;             /* указатель на следующую структуру */
   printf("him->favfood равно %s: him->handle.last равно %s\n",
       him->favfood, him->handle.last);

   return 0;
}
```

Запуск программы дает следующий вывод:

```
адрес #1: 0x7fff5fbff820 #2: 0x7fff5fbff874
указатель #1: 0x7fff5fbff820 #2: 0x7fff5fbff874
him->income равно $68112.00: (*him).income равно $68112.00
him->favfood равно рыбным фрикасе: him->handle.last равно Стюарт
```

Мы сначала посмотрим, каким образом создается указатель на структуру `guy`, а затем объясним, как описать отдельные члены структур с применением этого указателя.

Объявление и инициализация указателя на структуру

Объявить указатель на структуру очень просто:

```
struct guy * him;
```

Первым идет ключевое слово `struct`, затем дескриптор структуры `guy`, звездочка (`*`) и, наконец, имя указателя. Это тот же самый синтаксис, который используется для объявления других указателей, как было показано ранее.

Объявление не приводит к созданию новой структуры, но указатель `him` теперь может ссылаться на любую существующую структуру типа `guy`. Например, если `barney` – структура типа `guy`, то можно написать следующий оператор:

```
him = &barney;
```

В отличие от массивов, имя структуры не является ее адресом – вы должны применять операцию `&`.

В нашем примере `fellow` – массив структур, т.е. `fellow[0]` представляет собой структуру, поэтому код инициализирует `him`, делая его указывающим на `fellow[0]`:

```
him = &fellow[0];
```

Первые две строки вывода показывают, что присваивание прошло успешно. Сравнивая эти две строки, мы видим, что `him` указывает на `fellow[0]`, а `him + 1` – на `fellow[1]`. Обратите внимание, что добавление 1 к `him` приводит к добавлению значения 84 к адресу. В шестнадцатеричной форме записи 874 - 820 = 54 (шестнадцатеричное) = 84 (десятичное), т.к. каждая структура `guy` занимает 84 байта памяти: под `names.first` отводится 20 байтов, под `names.last` – 20 байтов, под `favfood` – 20 байтов, под `job` – 20 байтов и под `income` – 4 байта (размер типа `float` в нашей системе). Кстати, в некоторых системах размер структуры может быть больше суммы размеров ее частей. Причина в том, что требования к выравниванию данных системы могут вызвать появление зазоров. Например, возможно, что система должна размещать каждый член структуры по четному адресу либо по адресу, кратному 4. Такие структуры могут содержать в себе неиспользуемые "бреши".

Доступ к членам по указателю

Указатель `him` указывает на структуру `fellow[0]`. А как с помощью `him` получить значение члена `fellow[0]`? В третьей строке вывода демонстрируются два метода.

Первый и наиболее распространенный метод предусматривает применение новой операции `->`. Знак этой операции образован из дефиса (`-`) и символа "больше" (`>`). Мы имеем следующие зависимости:

```
him->income равно barney.income, если him == &barney
him->income равно fellow[0].income, если him == &fellow[0]
```

Другими словами, указатель на структуру, за которым следует операция `->`, работает таким же образом, как имя структуры с последующей операцией точки (`.`). (Вы не можете использовать просто `him.income`, потому что `him` не является именем структуры.)

Важно отметить, что `him` – указатель, но `him->income` – это член структуры, на которую он указывает. Это значит, что в данном случае `him->income` представляет собой переменную типа `float`.

Второй метод для указания значения члена структуры соответствует следующей последовательности утверждений: если `him == &fellow[0]`, то `*him == fellow[0]`, поскольку операции `&` и `*` являются обратными. Следовательно, после подстановки мы получаем такое выражение:

```
fellow[0].income == (*him).income
```

Круглые скобки здесь обязательны, т.к. операция . имеет более высокий приоритет, чем *. Подводя итоги, если him – это указатель на структуру типа guy по имени barney, то следующие выражения эквивалентны:

```
barney.income == (*him).income == him->income //предполагая, что him == &barney
```

А теперь давайте посмотрим на взаимодействие между структурами и функциями.

Сообщение функциям о структурах

Вспомните, что аргументы функции передают ей значения. Каждое значение является числом – возможно, int, float, ASCII-кодом символа или адресом.

Структура сложнее одиночного значения, поэтому не должно вызывать удивление то, что ранние реализации C не позволяют применять структуру в качестве аргумента для функции. В более новых реализациях это ограничение было снято, и ANSI C позволяет использовать структуры в аргументах функций. Таким образом, современные реализации C предлагают возможность выбора между передачей в качестве аргументов самих структур и указателей на эти структуры, либо, если вас интересует только часть структуры – передачей в аргументах членов структуры. Мы исследуем все три метода, начав с передачи членов структуры как аргументов.

Передача членов структуры

До тех пор, пока член структуры имеет тип данных с единственным значением (т.е. int или один из его производных типов, char, float, double либо указатель), его можно передавать в качестве аргумента функции, которая принимает этот конкретный тип. Простейшая программа финансового анализа из листинга 14.5, которая складывает сумму на обычном банковском счете клиента и сумму на его сберегательном счете, иллюстрирует это утверждение.

Листинг 14.5. Программа funds1.c

```
/* funds1.c -- передача членов структуры в качестве аргументов */
#include <stdio.h>
#define FUNDLEN 50
struct funds {
   char bank[FUNDLEN];
   double bankfund;
   char save[FUNDLEN];
   double savefund;
};
double sum(double, double);
int main(void)
{
   struct funds stan = {
      "Garlic-Melon Bank",
      4032.27,
      "Lucky's Savings and Loan",
      8543.94
   };
   printf("Общая сумма на счетах у Стэна составляет $%.2f.\n",
          sum(stan.bankfund, stan.savefund) );
   return 0;
}
/* суммирование двух чисел типа double */
double sum(double x, double y)
{
   return(x + y);
}
```

Вот результаты пробного запуска:

```
Общая сумма на счетах у Стэна составляет $12576.21.
```

Итак, программа работает. Обратите внимание, что функция sum() не знает, да и не заботится о том, являются ли фактические аргументы членами структуры; она только требует, чтобы они имели тип double.

Конечно, если вы хотите, чтобы вызываемая функция оказывала воздействие на значение члена в вызывающей функции, то можете передавать адрес этого члена:

```
modify(&stan.bankfund);
```

Это могла бы быть функция, изменяющая сумму на банковском счету Стэна.

Следующий подход к сообщению функции о структуре предусматривает уведомление о том, что функция имеет дело со структурой.

Использование адреса структуры

Мы будем решать ту же задачу, что и ранее, но на этот раз в качестве аргумента применим адрес структуры. Поскольку функция будет работать со структурой funds, она также должна использовать объявление funds. Код программы приведен в листинге 14.6.

Листинг 14.6. Программа funds2.c

```
/* funds2.c -- передача указателя на структуру */
#include <stdio.h>
#define FUNDLEN 50
struct funds {
   char bank[FUNDLEN];
   double bankfund;
   char save[FUNDLEN];
   double savefund;
};
double sum(const struct funds *); /* аргумент является указателем */
int main(void)
{
   struct funds stan = {
      "Garlic-Melon Bank",
      4032.27,
      "Lucky's Savings and Loan",
      8543.94
   };
   printf("Общая сумма на счетах у Стэна составляет $%.2f.\n", sum(&stan));
   return 0;
}
double sum(const struct funds * money)
{
   return(money->bankfund + money->savefund);
}
```

Запуск программы дает тот же самый результат:

```
Общая сумма на счетах у Стэна составляет $12576.21.
```

Функция sum() принимает указатель (money) на структуру funds в своем единственном аргументе. Передача адреса &stan функции приводит к тому, что теперь указатель money указывает на структуру stan. Затем с помощью операции -> мы получаем

значения stan.bankfund и stan.savefund. Поскольку функция не изменяет значение, на которое ссылается указатель, money объявляется как указатель на const.

Эта функция также имеет доступ к членам, представляющим названия учреждений, хотя и не пользуется ими. Обратите внимание, что для получения адреса структуры должна применяться операция &. В отличие от имени массива имя структуры не является синонимом ее адреса.

Передача структуры в качестве аргумента

Для компиляторов, которые разрешают передавать структуры в качестве аргументов, последний пример можно переписать так, как показано в листинге 14.7.

Листинг 14.7. Программа funds3.c

```
/* funds3.c -- передача структуры */
#include <stdio.h>
#define FUNDLEN 50

struct funds {
   char   bank[FUNDLEN];
   double bankfund;
   char   save[FUNDLEN];
   double savefund;
};

double sum(struct funds moolah); /* аргумент является структурой */

int main(void)
{
   struct funds stan = {
      "Garlic-Melon Bank",
      4032.27,
      "Lucky's Savings and Loan",
      8543.94
   };

   printf("Общая сумма на счетах у Стэна составляет $%.2f.\n", sum(&stan));

   return 0;
}

double sum(struct funds moolah)
{
   return(moolah.bankfund + moolah.savefund);
}
```

И снова вывод оказывается прежним:

```
Сумма на счету у Стэна составляет $12576.21.
```

Мы заменили указатель на struct funds по имени money переменной типа struct funds с именем moolah. При вызове sum() создается автоматическая переменная moolah, согласованная с шаблоном funds. Затем члены этой структуры инициализируются копиями значений соответствующих членов структуры stan. По этой причине вычисления производятся с участием копии исходной структуры, тогда как в предыдущей программе (в которой использовался указатель) задействована сама исходная структура. Так как moolah является структурой, в программе применяется moolah.bankfund, а не moolah->bankfund. С другой стороны, в листинге 14.6 используется money->bankfund, потому что money – указатель, а не структура.

Дополнительные возможности структур

Современный язык С позволяет присваивать одну структуру другой — то, чего нельзя делать с массивами. То есть, если n_data и o_data — структуры того же типа, то можно записать следующий код:

```
o_data = n_data;  // присваивание одной структуры другой
```

Это приводит к тому, что каждому члену n_data присваивается значение соответствующего члена o_data. Это работает, даже если член оказывается массивом. Кроме того, структуру можно инициализировать другой структурой того же типа:

```
struct names right_field = {"Джеймс", "Бонд"};
struct names captain = right_field; //инициализация структуры другой структурой
```

В современном языке С, включая ANSI С, структуры не только можно передавать функции в качестве аргументов, но также и возвращать их из функции. Применение структур в аргументах функции позволяет передавать ей информацию о структуре. Использование функций для возвращения структур дает возможность передавать информацию о структуре из вызываемой функции в вызывающую. Указатели на структуры также допускают двусторонний обмен данными, так что вы часто будете применять один из этих подходов при решении разнообразных задач. Рассмотрим еще один набор примеров, иллюстрирующих данные два подхода.

Чтобы сравнить эти два подхода, мы напишем простую программу, которая обрабатывает структуры с использованием указателей, и затем переделаем ее так, чтобы в ней выполнялась передача и возвращение структур. Сама программа запрашивает имя и фамилию и сообщает общее количество букв в них. Этот проект едва ли требует структур, но он предлагает простую инфраструктуру, которая позволяет увидеть, как они работают. В листинге 14.8 представлена версия программы с указателями.

Листинг 14.8. Программа names1.c

```
/* names1.c -- использует указатели на структуры */
#include <stdio.h>
#include <string.h>
#define NLEN 30
struct namect {
   char fname[NLEN];
   char lname[NLEN];
   int letters;
};
void getinfo(struct namect *);
void makeinfo(struct namect *);
void showinfo(const struct namect *);
char * s_gets(char * st, int n);
int main(void)
{
   struct namect person;
   getinfo(&person);
   makeinfo(&person);
   showinfo(&person);
   return 0;
}
void getinfo (struct namect * pst)
{
   printf("Введите свое имя.\n");
   s_gets(pst->fname, NLEN);
```

```
    printf("Введите свою фамилию.\n");
    s_gets(pst->lname, NLEN);
}
void makeinfo (struct namect * pst)
{
    pst->letters = strlen(pst->fname) +
    strlen(pst->lname);
}
void showinfo (const struct namect * pst)
{
    printf("%s %s, ваше имя и фамилия содержат %d букв.\n",
          pst->fname, pst->lname, pst->letters);
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');    // поиск новой строки
        if (find)                   // если адрес не равен NULL,
            *find = '\0';           // поместить туда нулевой символ
        else
            while (getchar() != '\n')
                continue;           // отбросить остаток строки
    }
    return ret_val;
}
```

Компиляция и запуск программы генерирует следующие результаты:

```
Введите свое имя.
Васисуалий
Введите свою фамилию.
Лоханкин
Васисуалий Лоханкин, ваше имя и фамилия содержат 18 букв.
```

Работа программы распределена между тремя функциями, вызываемыми в main(). В каждом случае функции передается адрес структуры person.

Функция getinfo() передает информацию изнутри себя в main(). В частности, она получает имена от пользователя и помещает их в структуру person, применяя для доступа к ней указатель pst. Вспомните, что pst->lname означает член lname структуры, на которую указывает pst. Это делает pst->lname эквивалентом имени массива значений char и, следовательно, подходящим аргументом для функции gets(). Обратите внимание, что хотя функция getinfo() снабжает информацией главную программу, она не использует для этого механизм возврата, поэтому имеет тип void.

Функция makeinfo() выполняет двустороннюю передачу информации. С применением указателя на person она находит имя и фамилию, хранящиеся в этой структуре. Она использует функцию strlen() из библиотеки C для подсчета количества букв в имени и фамилии, а затем применяет адрес структуры person для сохранения полученной суммы. Эта функция также имеет тип void. И, наконец, функция showinfo() использует указатель для доступа к информации, предназначенной для вывода. Поскольку showinfo() не изменяет содержимое массива, указатель объявлен как const.

Во всех этих операциях участвовала всего лишь одна переменная типа структуры `person`, и каждая из функций для доступа к структуре применяла ее адрес. Первая функция передавала информацию изнутри себя вызывающей программе, вторая функция принимала информацию из вызывающей программы внутрь себя, а третья функция делала то и другое.

Теперь посмотрим, каким образом запрограммировать решение той же задачи с использованием структур как аргументов и возвращаемых значений. Во-первых, для передачи самой структуры необходимо применять аргумент `person`, а не `&person`. Тогда соответствующий формальный аргумент объявляется с типом `struct namect`, а не указателем на этот тип. Во-вторых, чтобы предоставить `main()` значения структуры, можно возвратить саму структуру. В листинге 14.9 показана версия программы без указателей.

Листинг 14.9. Программа names2.c

```
/* names2.c -- передает и возвращает структуры */
#include <stdio.h>
#include <string.h>

#define NLEN 30
struct namect {
    char fname[NLEN];
    char lname[NLEN];
    int letters;
};

struct namect getinfo(void);
struct namect makeinfo(struct namect);
void showinfo(struct namect);
char * s_gets(char * st, int n);
int main(void)
{
    struct namect person;

    person = getinfo();
    person = makeinfo(person);
    showinfo(person);

    return 0;
}
struct namect getinfo(void)
{
    struct namect temp;
    printf("Введите свое имя.\n");
    s_gets(temp.fname, NLEN);
    printf("Введите свою фамилию.\n");
    s_gets(temp.lname, NLEN);
    return temp;
}
struct namect makeinfo(struct namect info)
{
    info.letters = strlen(info.fname) + strlen(info.lname);
    return info;
}
void showinfo(struct namect info)
{
    printf("%s %s, ваше имя и фамилия содержат %d букв.\n",
          info.fname, info.lname, info.letters);
}
```

```
char * s_gets(char * st, int n)
{
   char * ret_val;
   char * find;

   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      find = strchr(st, '\n');     // поиск новой строки
      if (find)                    // если адрес не равен NULL,
         *find = '\0';             // поместить туда нулевой символ
      else
         while (getchar() != '\n')
            continue;              // отбросить остаток строки
   }
   return ret_val;
}
```

Эта версия дает тот же самый результат, что и предыдущая, но работает по-другому. Каждая из трех функций создает собственную копию структуры person, так что в программе задействованы четыре разные структуры, а не только одна.

Для примера рассмотрим функцию makeinfo(). В первой программе ей передавался адрес структуры person, и функция имела дело с действительными значениями person. Во второй версии программы создавалась новая структура по имени info. Значения, хранящиеся в person, копировались в info, и функция работала с копией. Следовательно, подсчитанное количество букв сохранялось в info, но не в person. Тем не менее, это исправляет механизм возврата. Строка в makeinfo()

```
return info;
```

объединяется со строкой в main()

```
person = makeinfo(person);
```

для копирования значений хранящихся внутри info, в person. Обратите внимание, что функция makeinfo() должна быть объявлена с типом struct namect, т.к. она возвращает структуру.

Структуры или указатели на структуры?

Предположим, что вам необходимо написать функцию, связанную со структурами. Должны ли вы использовать указатели на структуры в качестве аргументов или же типы структур для аргументов и возвращаемых значений? Каждый подход характеризуется сильными и слабыми сторонами.

Метод с применением указателей в аргументах обладает двумя достоинствами: он работает как в старых, так и в новых реализациях C и является быстрым; передается всего лишь один адрес. Недостаток в том, что данные менее защищены. Некоторые операции в вызываемой функции могут непреднамеренно воздействовать на данные в исходной структуре. Однако появившийся в ANSI C спецификатор const решает эту проблему. Например, если в функцию showinfo() из листинга 11.8 поместить код, который изменяет какой-то член структуры, то компилятор обнаружит это и сообщит об ошибке.

Одно из преимуществ передачи структур в качестве аргументов заключается в том, что функция имеет дело с копией исходных данных, что безопаснее, чем работать с исходными данными. Вдобавок стиль программирования становится более ясным.

Предположим, что вы определили следующий тип структуры:

```
struct vector {double x; double y;};
```

Вы хотите установить вектор ans в сумму векторов a и b. Вы могли бы написать функцию, передающую и возвращающую структуры, которая привела бы к следующему коду:

```
struct vector ans, a, b;
struct vector sum_vect(struct vector, struct vector);
...
ans = sum_vect(a,b);
```

Для инженера эта версия выглядит более естественной, чем версия с указателями, которая могла бы выглядеть так:

```
struct vector ans, a, b;
void sum_vect(const struct vector *, const struct vector *, struct vector *);
...
sum_vect(&a, &b, &ans);
```

Кроме того, в версии с указателями пользователь должен помнить, каким аргументом должен быть представлен адрес суммы — первым или последним.

Два основных недостатка передачи структур связаны с тем, что старые реализации С могут не воспринимать такой код, а также с тем, что в этом случае неэкономно расходуется время и память. Особенно расточительно передавать крупные структуры в функцию, которая использует только один или два члена структуры. В этом случае имеет смысл передавать указатель или же требуемые члены как индивидуальные аргументы.

Обычно программисты применяют указатели на структуры в качестве аргументов функции из соображений эффективности, используя const, когда необходимо защитить данные от нежелательных изменений. Передача структур по значению чаще всего делается для структур небольших размеров.

Символьные массивы или указатели на char в структурах

В рассмотренных ранее примерах для хранения строк в структуре применялись символьные массивы. Вероятно, вас интересует, можно ли вместо них использовать указатели на char? Например, в листинге 14.3 имеется следующее объявление:

```
#define LEN 20
struct names {
   char first[LEN];
   char last[LEN];
};
```

Можно ли вместо этого поступить так?

```
struct pnames {
   char * first;
   char * last;
};
```

Ответ — да, это возможно, но могут возникнуть проблемы, если вы не обдумаете все последствия. Взгляните на показанный ниже код:

```
struct names veep = {"Talia", "Summers"};
struct pnames treas = {"Brad", "Fallingjaw"};
printf("%s и %s\n", veep.first, treas.first);
```

Этот код допустим, и он работает, однако рассмотрим, где хранятся строки. В случае переменной veep типа struct names строки хранятся внутри структуры; для хранения двух имен структура выделяет всего 40 байтов. Тем не менее, в переменной treas типа struct pnames строки хранятся там, где компилятор сохраняет строковые константы. Все, что содержит данная структура — это два адреса, которые в нашей системе в целом занимают 16 байтов. В частности, структура struct pnames не выделяет память для хранения строк. Она может применяться только со строками, для которых память была выделена где-то в другом месте, такими как строковые константы или строки в массивах. Короче говоря, указатели в структуре struct pnames должны использоваться только для управления строками, которые были созданы с выделением под них памяти в другом месте программы.

Давайте посмотрим, когда это ограничение превращается в проблему. Взгляните на следующий код:

```
struct names accountant;
struct pnames attorney;
puts("Введите фамилию вашего бухгалтера:");
scanf("%s", accountant.last);
puts("Введите фамилию вашего адвоката:");
scanf("%s", attorney.last);  /* здесь скрыта опасность */
```

С точки зрения синтаксиса этот код допустим. Но куда сохраняются входные данные? Фамилия бухгалтера записывается в последний член переменной accountant; эта структура содержит массив для хранения строки. В случае фамилии адвоката функция scanf() получает указание поместить строку фамилии по адресу, заданному как attorney.last. Из-за того, что эта переменная не инициализирована, адрес может иметь произвольное значение, и программа может попытаться поместить фамилию куда угодно. Если повезет, то программа будет работать, по крайней мере, некоторое время, либо сразу же аварийно завершится. Однако если программа работает, то вам на самом деле не повезло, т.к. в ней присутствует катастрофическая ошибка, о которой вы не знаете.

Таким образом, если вам необходима структура для хранения строк, то проще применять члены типа символьных массивов. Использование указателей на char в отдельных случаях допускается, но потенциально сопряжено с серьезными проблемами.

Структура, указатели и malloc()

Применение в структуре указателя, поддерживающего строку, имеет смысл, когда с помощью функции malloc() для строки выделяется область памяти и указатель используется, чтобы сохранить адрес этой области. Преимущество такого подхода заключается в том, что malloc() позволяет выделить ровно столько памяти, сколько необходимо для строки. Вы можете запросить 4 байта для сохранения строки "Joe" и 18 байтов для строки с мадагаскарским именем "Rasolofomasoandro". Код в листинге 14.9 совсем нетрудно адаптировать под этот подход. Два основных изменения касаются определения структуры с целью применения указателей вместо массивов и предоставления новой версии функции getinfo().

Новое определение структуры будет выглядеть следующим образом:

```
struct namect {
    char * fname;  // использование указателей вместо массивов
    char * lname;
    int letters;
};
```

Новая версия getinfo() будет читать входные данные во временный массив, использовать malloc() для выделения пространства памяти и копировать в него строку. Она будет делать это для каждого имени:

```
void getinfo (struct namect * pst)
{
    char temp[SLEN];
    printf("Введите свое имя.\n");
    s_gets(temp, SLEN);
    // выделение памяти для хранения имени
    pst->fname = (char *) malloc(strlen(temp) + 1);
    // копирование имени в выделенную память
    strcpy(pst->fname, temp);
    printf("Введите свою фамилию.\n");
    s_gets(temp, SLEN);
    pst->lname = (char *) malloc(strlen(temp) + 1);
    strcpy(pst->lname, temp);
}
```

Вы должны четко понимать, что эти две строки не хранятся в структуре. Они сохранены в области памяти, управляемой malloc(). Тем не менее, адреса двух строк хранятся в структуре, и именно с ними обычно имеют дело функции обработки строк. Следовательно, остальные функции в программе в изменениях не нуждаются.

Однако, согласно совету из главы 12, вы должны уравновешивать вызовы malloc() вызовами free(), поэтому в программу добавлена новая функция по имени cleanup(), которая освобождает память, когда программа завершает пользоваться ею. Вы найдете эту новую функцию и оставшуюся часть программы в листинге 14.10.

Листинг 14.10. Программа names3.c

```
// names3.c -- использование указателей и функции malloc()
#include <stdio.h>
#include <string.h>      // для strcpy(), strlen()
#include <stdlib.h>      // для malloc(), free()
#define SLEN 81

struct namect {
    char * fname;        // использование указателей
    char * lname;
    int letters;
};

void getinfo(struct namect *);      // выделение памяти
void makeinfo(struct namect *);
void showinfo(const struct namect *);
void cleanup(struct namect *);      // освобождение памяти, когда она больше не нужна
char * s_gets(char * st, int n);

int main(void)
{
    struct namect person;

    getinfo(&person);
    makeinfo(&person);
    showinfo(&person);
    cleanup(&person);

    return 0;
}
```

```
void getinfo (struct namect * pst)
{
   char temp[SLEN];
   printf("Введите свое имя.\n");
   s_gets(temp, SLEN);
   // выделение памяти для хранения имени
   pst->fname = (char *) malloc(strlen(temp) + 1);
   // копирование имени в выделенную память
   strcpy(pst->fname, temp);
   printf("Введите свою фамилию.\n");
   s_gets(temp, SLEN);
   pst->lname = (char *) malloc(strlen(temp) + 1);
   strcpy(pst->lname, temp);
}
void makeinfo (struct namect * pst)
{
   pst->letters = strlen(pst->fname) +
   strlen(pst->lname);
}
void showinfo (const struct namect * pst)
{
   printf("%s %s, ваше имя и фамилия содержат %d букв.\n",
   pst->fname, pst->lname, pst->letters);
}
void cleanup(struct namect * pst)
{
   free(pst->fname);
   free(pst->lname);
}
char * s_gets(char * st, int n)
{
   char * ret_val;
   char * find;
   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      find = strchr(st, '\n');     // поиск новой строки
      if (find)                    // если адрес не равен NULL,
         *find = '\0';             // поместить туда нулевой символ
      else
         while (getchar() != '\n')
            continue;              // отбросить остаток строки
   }
   return ret_val;
}
```

Ниже показан пример вывода:

```
Введите свое имя.
Васисуалий
Введите свою фамилию.
Лоханкин
Васисуалий Лоханкин, ваше имя и фамилия содержат 18 букв.
```

Составные литералы и структуры (C99)

Средство составных литералов C99 доступно для структур, а также для массивов. Оно удобно, если требуется всего лишь временное значение структуры. Например, составные литералы можно применять для создания структуры, предназначенной для использования в качестве аргумента функции или для присваивания другой структуре. Синтаксис составного литерала выглядит как заключенный в фигурные скобки список инициализаторов, которому предшествует имя типа в круглых скобках. Ниже представлен составной литерал типа struct book:

```
(struct book) {"Идиот", "Федор Достоевский", 6.99}
```

В листинге 14.11 приведен пример применения составных литералов для предоставления двух альтернативных значений переменной структуры. (На момент написания книги это средство поддерживалось не всеми компиляторами, но со временем ситуация должна поменяться.)

Листинг 14.11. Программа complit.c

```
/* complit.c -- составные литералы */
#include <stdio.h>
#define MAXTITL 41
#define MAXAUTL 31

struct book {          // шаблон структуры: book - дескриптор
   char title[MAXTITL];
   char author[MAXAUTL];
   float value;
};
int main(void)
{
   struct book readfirst;
   int score;
   printf("Введите рейтинг: ");
   scanf("%d",&score);
   if(score >= 84)
      readfirst = (struct book) {"Преступление и наказание",
                                 "Федор Достоевский",
                                 11.25};
   else
      readfirst = (struct book) {"Красивая шляпа мистера Баунси",
                                 "Фред Уинсом",
                                 5.99};
   printf("Назначенные вами рейтинги:\n");
   printf("%s by %s: $%.2f\n",readfirst.title,
        readfirst.author, readfirst.value);
   return 0;
}
```

Составные литералы можно использовать также как аргументы функций. Если функция ожидает структуру, то ей можно передавать в качестве фактического аргумента составной литерал:

```
struct rect {double x; double y;};
double rect_area(struct rect r){return r.x * r.y;}
...
double area;
area = rect_area( (struct rect) {10.5, 20.0});
```

Это приводит к тому, что area присваивается значение 210.0.

Если функция ожидает адрес, то ей можно передать адрес составного литерала:

```
struct rect {double x; double y;};
double rect_areap(struct rect * rp){return rp->x * rp->y;}
...
double area;
area = rect_areap( &(struct rect) {10.5, 20.0});
```

В результате переменная area получает значение 210.0.

Составные литералы, которые встречаются за пределами любых функций, имеют статическую продолжительность хранения, а те, что находятся внутри блока – автоматическую продолжительность хранения. В отношении составных литералов действуют те же синтаксические правила, что и для обычных списков инициализаторов. Это значит, например, что в составных литералах можно применять назначенные инициализаторы.

Члены с типами гибких массивов (C99)

В стандарте C99 предлагается новое средство, которое называется *членом типа гибкого массива*. Оно позволяет объявлять структуру, последний член в которой является массивом со специальными свойствами. Одно из специальных свойств заключается в том, что такой массив не существует – во всяком случае, не появляется немедленно. Второе специальное свойство состоит в том, что при наличии корректного кода член типа гибкого массива можно использовать, как если бы он существовал и имел нужное количество элементов. Возможно, это звучит несколько своеобразно, так что давайте рассмотрим шаги по созданию и применению структуры с членом типа гибкого массива.

Для начала ниже представлены правила, регламентирующие создание члена типа гибкого массива.

- Член типа гибкого массива должен быть последним в структуре.
- В структуре должен присутствовать, по крайней мере, еще один член другого типа.
- Гибкий массив объявляется подобно обычному массиву, но с пустыми квадратными скобками.

Вот пример, иллюстрирующий эти правила:

```
struct flex
{
    int count;
    double average;
    double scores[];  // член типа гибкого массива
};
```

Если вы объявили переменную типа struct flex, то не можете использовать член scores, т.к. память для него не зарезервирована. На самом деле, даже не подразумевается, что вы будете объявлять переменные типа struct flex. Вместо этого предполагается, что вы объявите *указатель* на тип struct flex, а затем с помощью malloc() выделите область памяти, достаточную для хранения обычного содержимого struct flex, *плюс* дополнительное пространство, которое необходимо для члена с типом гибкого массива. Например, пусть вы хотите, чтобы член scores представлял массив из пяти значений double. В этом случае понадобится поступить так:

```
struct flex * pf;  // объявление указателя
```

```
// запрос области памяти для размещения структуры и массива
pf = malloc(sizeof(struct flex) + 5 * sizeof(double));
```

Теперь вы располагаете объемом памяти, которого достаточно для хранения count, average и массива из пяти значений double. Для доступа к этим членам можно применять указатель pf:

```
pf->count = 5;           // установка члена count
pf->scores[2] = 18.5;    // доступ к элементу члена типа массива
```

В листинге 14.12 пример продолжает развиваться; гибкий массив получает возможность представлять пять значений в первом случае и девять значений — во втором. Здесь также демонстрируется написание функции для обработки структуры с членом типа гибкого массива.

Листинг 14.12. Программа flexmemb.c

```
// flexmemb.c -- член типа гибкого массива (средство C99)
#include <stdio.h>
#include <stdlib.h>

struct flex
{
   size_t count;
   double average;
   double scores[];   // член с типом гибкого массива
};
void showFlex(const struct flex * p);
int main(void)
{
   struct flex * pf1, *pf2;
   int n = 5;
   int i;
   int tot = 0;

   // выделение памяти для структуры и массива
   pf1 = malloc(sizeof(struct flex) + n * sizeof(double));
   pf1->count = n;
   for (i = 0; i < n; i++)
   {
      pf1->scores[i] = 20.0 - i;
      tot += pf1->scores[i];
   }
   pf1->average = tot / n;
   showFlex(pf1);
   n = 9;
   tot = 0;
   pf2 = malloc(sizeof(struct flex) + n * sizeof(double));
   pf2->count = n;
   for (i = 0; i < n; i++)
   {
      pf2->scores[i] = 20.0 - i/2.0;
      tot += pf2->scores[i];
   }
   pf2->average = tot / n;
   showFlex(pf2);
   free(pf1);
   free(pf2);
   return 0;
}
```

```
void showFlex(const struct flex * p)
{
   int i;
   printf("Рейтинги: ");
   for (i = 0; i < p->count; i++)
      printf("%g ", p->scores[i]);
   printf("\nСреднее значение: %g\n", p->average);
}
```

Ниже показан вывод:

```
Рейтинги: 20 19 18 17 16
Среднее значение: 18
Рейтинги: 20 19.5 19 18.5 18 17.5 17 16.5 16
Среднее значение: 17
```

К обработке структур, содержащих члены с типами гибких массивов, предъявляется ряд специальных требований. Во-первых, не используйте присваивание структур для копирования:

```
struct flex * pf1, *pf2;    // *pf1 и *pf2 являются структурами
...
*pf2 = *pf1;                // не поступайте так
```

Такой код привел бы к копированию только членов структуры, которые не относятся к типу гибкого массива. Вместо этого применяйте функцию `memcpy()`, которая описана в главе 16.

Во-вторых, не используйте такие структуры совместно с функциями, которые передают структуры по значению. Причина такого ограничения та же – передача аргумента по значению подобна присваиванию. Вместо этого применяйте функции, которые передают адрес структуры.

В-третьих, не используйте структуру с членом типа гибкого массива в качестве элемента массива или члена другой структуры.

Возможно, вы уже слышали о конструкции, подобной члену типа гибкого массива, которая называется *приемом "struct hack"*. Вместо применения пустых квадратных скобок для объявления члена типа гибкого массива прием "struct hack" предусматривает указание нулевого размера массива. Однако данный прием работал только с конкретным компилятором (GCC); он не входил в стандарт С. Подход с использованием члена типа гибкого массива предлагает методику, одобренную стандартом.

Анонимные структуры (C11)

Анонимная структура – это член структуры, который является неименованной структурой. Чтобы посмотреть, как это работает, сначала рассмотрим следующее определение для вложенной структуры:

```
struct names
{
   char first[20];
   char last[20];
};
struct person
{
   int id;
   struct names name;  // член, представляющий собой вложенную структуру
};
struct person ted = {8483, {"Ted", "Grass"}};
```

В этом примере член name – это вложенная структура, и для получения доступа к "Ted" можно было бы применить выражение наподобие ted.name.first:

```
puts(ted.name.first);
```

Стандарт C11 позволяет определять структуру person, используя в качестве члена вложенную неименованную структуру:

```
struct person
{
   int id;
   struct {char first[20]; char last[20];};  // анонимная структура
};
```

Эту структуру можно было бы инициализировать в той же манере:

```
struct person ted = {8483, {"Ted", "Grass"}};
```

Но доступ к членам упрощается, поскольку для этого применяются имена членов вроде first, как если бы они были членами person:

```
puts(ted.first);
```

Разумеется, можно было бы просто сделать first и last непосредственными членами структуры person, избавившись от вложенной структуры. Средство анонимности более полезно с вложенными объединениями, которые будут обсуждаться далее в главе.

Функции, использующие массив структур

Предположим, что имеется массив структур, который необходимо обработать с помощью функции. Имя массива – это синоним его адреса, так что его можно передавать функции. Вдобавок функция нуждается в доступе к шаблону структуры. Чтобы продемонстрировать, как это работает, в листинге 14.13 программа финансового анализа расширена с целью обслуживания двух человек, поэтому в ней присутствует массив из двух структур funds.

Листинг 14.13. Программа funds4.c

```
/* funds4.c -- передача функции массива структур */
#include <stdio.h>
#define FUNDLEN 50
#define N 2

struct funds {
   char bank[FUNDLEN];
   double bankfund;
   char save[FUNDLEN];
   double savefund;
};

double sum(const struct funds money[], int n);
int main(void)
{
   struct funds jones[N] = {
      {
         "Garlic-Melon Bank",
         4032.27,
         "Lucky's Savings and Loan",
         8543.94
      },
```

```
      {
         "Honest Jack's Bank",
         3620.88,
         "Party Time Savings",
         3802.91
      }
   };
   printf("Общая сумма на счетах у Джонсов составляет $%.2f.\n",
          sum(jones,N));

   return 0;
}
double sum(const struct funds money[], int n)
{
   double total;
   int i;

   for (i = 0, total = 0; i < n; i++)
      total += money[i].bankfund + money[i].savefund;

   return(total);
}
```

Вот вывод программы:

```
Общая сумма на счетах у Джонсов составляет $20000.00.
```

Имя массива jones является его адресом. В частности, это адрес первого элемента массива, которым представляет собой структуру jones[0]. Таким образом, первоначально указатель money задается следующим выражением:

```
money = &jones[0];
```

Поскольку money указывает на первый элемент массива jones, то money[0] — это еще одно имя первого элемента массива. Аналогично, money[1] — второй элемент массива. Каждый элемент является структурой funds, поэтому для каждого из них можно применять операцию точки (.), чтобы обращаться к членам структуры.

Ниже перечислены основные аспекты.

- Имя массива можно использовать для передачи в функцию адреса первой структуры массива.
- Для доступа к последующим структурам массива можно применять запись с квадратными скобками. Обратите внимание, что вызов функции

  ```
  sum(&jones[0], N)
  ```

  приведет к таким же результатам, как и в случае указания имени массива, поскольку jones и &jones[0] — это один и тот же адрес. Использование имени массива представляет собой просто косвенный способ передачи адреса структуры.
- Из-за того, что функция sum() не должна изменять исходные данные, в ней применяется квалификатор const из ANSI C.

Сохранение содержимого структур в файле

Поскольку структуры могут содержать самую разнообразную информацию, они являются важными инструментами для построения баз данных. Например, структуру можно использовать для хранения информации о служащих компании или об автомобильных запчастях. В итоге неизбежно возникнет необходимость сохранять эту

информацию в файле и извлекать ее из файла. Файл базы данных может содержать произвольное количество таких объектов данных. Полный набор информации, хранящейся в структуре, называется *записью*, а отдельные члены структуры – *полями*. Давайте рассмотрим эту тему более подробно.

Вероятно наиболее очевидный, но и наименее эффективный способ сохранения записи предполагает применение функции `fprintf()`.

В качестве примера вспомним структуру `book`, определенную в листинге 14.1:

```
#define MAXTITL 40
#define MAXAUTL 40
struct book {
   char title[MAXTITL];
   char author[MAXAUTL];
   float value;
};
```

Если `pbooks` идентифицирует файловый поток, то информацию из переменной `primer` типа `struct book` можно было бы сохранить с помощью следующего оператора:

```
fprintf(pbooks, "%s %s %.2f\n", primer.title,
        primer.author, primer.value);
```

Такой подход становится громоздким для структур, которые имеют, скажем, 30 членов. Кроме того, возникает проблема извлечения, т.к. программе необходим какой-то способ выяснения, где одно поле заканчивается, а другое начинается. Проблему можно решить, используя формат с полями фиксированного размера (например, `"%39s%39s%8.2f"`), но громоздкость никуда не девается.

Более приемлемое решение заключается в применении функций `fread()` и `fwrite()` для чтения и записи единиц с размером структуры. Вспомните, что эти функции производят чтение и запись с использованием такого же двоичного представления, как и программа. Например, вызов

```
fwrite(&primer, sizeof (struct book), 1, pbooks);
```

переходит к начальному адресу структуры `primer` и копирует все байты этой структуры в файл, ассоциированный с `pbooks`. Выражение `sizeof (struct book)` сообщает функции размер блока, подлежащего копированию, а 1 означает, что должен копироваться только один блок. Функция `fread()` с теми же аргументами копирует порцию данных размером со структуру из файла в область памяти, на которую указывает `&primer`. Короче говоря, эти функции читают и записывают за один раз полную запись, а не поле.

Один из недостатков хранения данных в двоичном представлении связан с тем, что в разных системах могут применяться отличающиеся двоичные представления, поэтому файл данных может оказаться непереносимым. Даже в одной и той же системе разные настройки компилятора могут в результате приводить к получению разных двоичных представлений.

Пример сохранения структуры

Чтобы продемонстрировать использование этих функций в программе, мы модифицировали код из листинга 14.2, чтобы сведения о книгах сохранялись в файле по имени `book.dat`. Если файл уже существует, программа отображает его текущее содержимое и затем позволяет добавить в файл новые данные. Новая версия программы показана в листинге 14.14. (Если вы имеете дело с Borland C/C++, ознакомьтесь с врезкой "Borland C и плавающая запятая" ранее в главе.)

Листинг 14.14. Программа booksave.c

```
/* booksave.c -- сохранение содержимого структуры в файле */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAXTITL 40
#define MAXAUTL 40
#define MAXBKS 10             /* максимальное количество книг */
char * s_gets(char * st, int n);
struct book {                /* определение шаблона book */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};

int main(void)
{
    struct book library[MAXBKS]; /* массив структур */
    int count = 0;
    int index, filecount;
    FILE * pbooks;
    int size = sizeof (struct book);
    if ((pbooks = fopen("book.dat", "a+b")) == NULL)
    {
        fputs("Не удается открыть файл book.dat\n",stderr);
        exit(1);
    }

    rewind(pbooks);           /* переход в начало файла */
    while (count < MAXBKS && fread(&library[count], size,
                               1, pbooks) == 1)
    {
        if (count == 0)
            puts("Текущее содержимое файла book.dat:");
        printf("%s авторства %s: $%.2f\n",library[count].title,
               library[count].author, library[count].value);
        count++;
    }
    filecount = count;
    if (count == MAXBKS)
    {
        fputs("Файл book.dat заполнен.", stderr);
        exit(2);
    }

    puts("Введите названия новых книг.");
    puts("Нажмите [enter] в начале строки, чтобы закончить ввод.");
    while (count < MAXBKS && s_gets(library[count].title, MAXTITL) != NULL
          && library[count].title[0] != '\0')
    {
        puts("Теперь введите имя автора.");
        s_gets(library[count].author, MAXAUTL);
        puts("Теперь введите цену книги.");
        scanf("%f", &library[count++].value);
        while (getchar() != '\n')
            continue;          /* очистить входную строку */
        if (count < MAXBKS)
            puts("Введите название следующей книги.");
    }
```

```
   if (count > 0)
   {
      puts("Каталог ваших книг:");
      for (index = 0; index < count; index++)
         printf("%s авторства %s: $%.2f\n",library[index].title,
                library[index].author, library[index].value);
      fwrite(&library[filecount], size, count - filecount,
               pbooks);
   }
   else
      puts("Вообще нет книг? Очень плохо.\n");

   puts("Программа завершена.\n");
   fclose(pbooks);

   return 0;
}

char * s_gets(char * st, int n)
{
   char * ret_val;
   char * find;

   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      find = strchr(st, '\n');     // поиск новой строки
      if (find)                    // если адрес не равен NULL,
         *find = '\0';             // поместить туда нулевой символ
      else
         while (getchar() != '\n')
            continue;              // отбросить остаток строки
   }
   return ret_val;
}
```

Мы сначала посмотрим на результаты двух пробных запусков и затем обсудим основные особенности программы.

```
$ booksave
Введ$ите названия новых книг.
Нажмите [enter] в начале строки, чтобы закончить ввод.
Metric Merriment
Теперь введите имя автора.
Polly Poetica
Теперь введите цену.
18.99
Введите название следующей книги.
Deadly Farce
Теперь введите имя автора.
Dudley Forse
Теперь введите цену.
15.99
Введите название следующей книги.
[enter]
Каталог ваших книг:
Metric Merriment авторства Polly Poetica: $18.99
Deadly Farce авторства Dudley Forse: $15.99
Программа завершена.
```

```
$ booksave
Текущее содержимое файла book.dat:
Metric Merriment авторства Polly Poetica: $18.99
Deadly Farce авторства Dudley Forse: $15.99
Введите названия новых книг.
The Third Jar
Теперь введите имя автора.
Nellie Nostrum
Теперь введите цену.
22.99
Введите название следующей книги.
[enter]
Каталог ваших книг:
Metric Merriment авторства Polly Poetica: $18.99
Deadly Farce авторства Dudley Forse: $15.99
The Third Jar авторства Nellie Nostrum: $22.99
Программа завершена.
$
```

При следующем запуске программы booksave.c все три книги будут отображены как текущие записи файла book.dat.

Анализ программы

Для начала файл открывается в режиме "a+b". Часть a+ позволяет программе читать весь файл и добавлять данные в конец файла. Часть b — это принятый в ANSI способ для сообщения о том, что программа будет применять двоичный файловый формат. Для систем Unix, которые не принимают часть b, ее можно опустить, поскольку Unix в любом случае имеется только одна форма файлов. Для реализаций C, предшествующих ANSI, может понадобиться найти локальный эквивалент b.

Двоичный режим был выбран из-за того, что функции fread() и fwrite() предназначены для работы с двоичными файлами. Действительно, некоторое содержимое структуры является текстовым, однако член value — нет. Если вы воспользуетесь текстовым редактором для просмотра файла book.dat, то текстовая часть будет отображаться нормально, но числовая часть окажется нечитабельной и может даже стать причиной выдачи предупреждений.

Вызов rewind() обеспечивает установку указателя позиции в файле на начало файла, приводя его в состояние готовности к первому чтению.

Первый цикл while читает одну структуру за раз в массив структур, останавливаясь при заполнении этого массива либо при исчерпании данных в файле. Переменная filecount отслуживает количество прочитанных структур.

Следующий цикл while запрашивает и получает пользовательский ввод. Как и в листинге 14.2, этот цикл прекращается, когда массив заполнен или пользователь нажал клавишу <Enter> в начале строки. Обратите внимание, что переменная count начинается со значения, которое она получила по окончании предыдущего цикла. Это приводит к добавлению новых записей в конец массива.

Затем в цикле for выводятся данные, полученные из файла и от пользователя. Так как файл был открыт в режиме добавления, новые записи присоединяются к существующему содержимому.

Мы могли бы воспользоваться циклом и добавлять структуры в конец файла по одной за раз. Однако мы решили прибегнуть к способности функции fwrite() записывать более одного блока за раз. Выражение count - filecount дает количество добавленных новых книг, а вызов fwrite() записывает в файл такое количество блоков

размером со структуру. Выражение `&library[filecount]` – это адрес первой новой структуры в массиве, поэтому копирование начинается с этой точки.

Вероятно, рассмотренный пример является простейшим способом записи структур в файл и их извлечения из него, но в нем в нем может понапрасну расходоваться пространство, поскольку также сохраняются и неиспользуемые части структуры.

Размер структуры составляет `2 × 40 × sizeof (char) + sizeof (float)`, что в нашей системе дает в сумме 84 байта. Ни одна из записей в действительности не требует всего этого пространства. Тем не менее, одинаковый размер всех порций данных упрощает извлечение данных.

Другой подход заключается в применении записей переменных размеров. Для облегчения считывания таких записей из файла каждая запись может начинаться с числового поля, указывающего размер записи. Это немного сложнее того, что мы только что делали. Обычно данный подход предусматривает использование связных структур, которые мы исследуем далее, и динамического выделения памяти, обсуждаемого в главе 16.

Структуры: что дальше?

Прежде чем завершить исследование структур, мы хотели бы упомянуть одно из наиболее важных применений структур – создание новых форм данных. Для решения определенных задач пользователям компьютера необходимы намного более эффективные формы данных, чем простые массивы и структуры, которые были представлены ранее. Формы данных подобного рода получили собственные названия, такие как очереди, двоичные деревья, кучи, хеш-таблицы и графы. Многие формы построены на основе связных структур. Обычно каждая структура содержит один или два элемента данных плюс один или два указателя на другие структуры того же типа. Эти указатели связывают одну структуру с другой и образуют путь, позволяющий выполнить проход по всей совокупности структур. Например, на рис. 14.3 представлена структура двоичного дерева, где каждая индивидуальная структура (или узел) соединена с двумя структурами уровнем ниже.

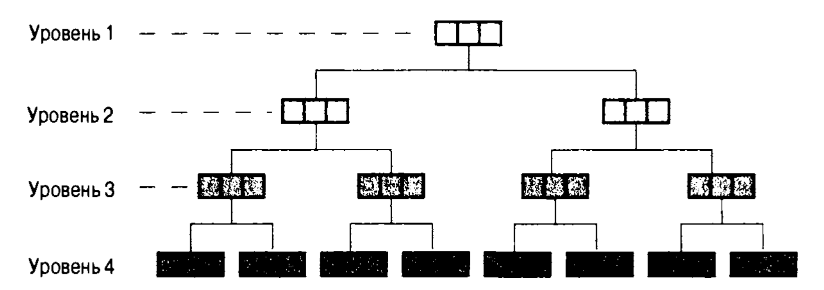


Рис. 14.3. Структура двоичного дерева

Является ли иерархическая, или *древовидная*, структура, показанная на рис. 14.3, более эффективной, чем массив? Рассмотрим случай дерева с 10 уровнями узлов. Оно имеет $2^{10} - 1$, или 1023, узла, в которых можно было бы хранить вплоть до 1023 слов. Если эти слова упорядочены в соответствии с некоторым осмысленным планом, то нужное слово можно найти максимум за девять перемещений по мере того, как поиск продвигается вниз с одного уровня на следующий. Если бы слова хранились в массиве, то для нахождения искомого слова, в худшем случае пришлось бы просмотреть все 1023 элемента.

Если вас интересуют более сложные концепции подобного рода, можете обратиться к разнообразным научным публикациям, которые посвящены структурам данных. С помощью структур С можно создавать и пользоваться практически любой формой данных из числа представленных в этих публикациях. Кроме того, некоторые сложные формы исследуются в главе 17.

На этом мы завершаем обзор структур в настоящей главе, но еще приведем примеры связных структур в главе 17. Далее мы обсудим три других средства С для работы с данными: объединения, перечисления и typedef.

Объединения: краткое знакомство

Объединение – это тип, который позволяет хранить данные разных типов в одном и том же месте памяти (но не одновременно). Типичным видом объединения может служить таблица, предназначенная для хранения смеси типов в определенном порядке, который не является ни регулярным, ни известным заранее. Применяя массив объединений, можно создать массив единиц одинаковых размеров, каждая из которых может содержать данные разнообразных типов.

Объединения формируются во многом подобно структурам. Имеется шаблон объединения и переменная типа объединения. Они могут быть определены с помощью одного действия или двух за счет использования дескриптора объединения. Ниже показан пример шаблона объединения с дескриптором:

```
union hold {
   int digit;
   double bigfl;
   char letter;
};
```

Структура с похожим объявлением способна хранить значения типов int, double *и* char одновременно, однако объединение может хранить значение типа int *или* double *или* char. Вот пример определения трех переменных объединения типа hold:

```
union hold fit;          // переменная объединения типа hold
union hold save[10];     // массив из 10 переменных объединения
union hold * pu;         // указатель на переменную типа hold
```

Первое объявление создает одиночную переменную fit. Компилятор выделяет пространство памяти, достаточное для хранения наибольшего из описанных возможностей. В данном случае наибольшим вариантом из перечисленных является тип double, который в нашей системе требует 64 бита, или 8 байтов. Второе объявление создает массив по имени save с 10 элементами, каждый из которых имеет размер 8 байтов. Третье объявление создает указатель, который может содержать адрес объединения hold.

Объединение можно инициализировать. Поскольку объединение хранит только одно значение, правила его инициализации отличаются от таких правил для структур. В частности, вам доступны три варианта: инициализировать объединение другим объединением того же типа, инициализировать первый элемент объединения или в случае С99 применить назначенный инициализатор:

```
union hold valA;
valA.letter = 'R';
union hold valB = valA;  // инициализация одного объединения другим
union hold valC = {88};  // инициализация члена digit объединения
union hold valD = {.bigfl = 118.2};  // назначенный инициализатор
```

Использование объединений

Ниже показано, как можно использовать объединение:

```
fit.digit = 23;      // в переменной fit хранится 23; используются 2 байта
fit.bigfl = 2.0;     // 23 очищено, 2.0 сохранено; используются 8 байтов
fit.letter = 'h';    // 2.0 Значение, h сохранено; используется 1 байт
```

Операция точки показывает, какой тип данных применяется в текущий момент. За один раз сохраняется только одно значение. Нельзя одновременно хранить значение char и int, несмотря на то, что пространства для этого вполне достаточно. Ответственность за отслеживание в программе типа данных, хранящегося в текущий момент внутри объединения, возлагается на вас.

Вы можете использовать операцию -> с указателями на объединения в той же манере, как применяли ее с указателями на структуры:

```
pu = &fit;
x = pu->digit;       // то же, что и x = fit.digit
```

Ниже показано, как *не* следует поступать:

```
fit.letter = 'A';
flnum = 3.02*fit.bigfl; // ОШИБКА!
```

Эта последовательность ошибочна, т.к. сохранено значение типа char, но в следующей строке предполагается, что содержимое fit имеет тип double.

Тем не менее, иногда бывает полезно использовать один член для помещения значений в объединение, а другой — для просмотра содержимого объединения. В листинге 15.4 в следующей главе показан пример.

Другой ситуацией применения объединений является структура, в которой сохраняемая информация зависит от значения одного из ее членов. Предположим, что у вас есть структура, представляющая автомобиль. Если автомобиль принадлежит пользователю, вы хотите, чтобы член структуры описывал владельца. Если автомобиль взят напрокат, необходимо, чтобы член описывал компанию по прокату. Тогда можно записать так:

```
struct owner {
   char socsecurity[12];
   ...
};

struct leasecompany {
   char name[40];
   char headquarters[40];
   ...
};

union data {
   struct owner owncar;
   struct leasecompany leasecar;
};

struct car_data {
   char make[15];
   int status;  /* 0 = принадлежит, 1 = взят напрокат */
   union data ownerinfo;
   ...
};
```

Пусть flits – это структура car_data. Тогда если значение flits.status равно 0, программа может использовать flits.ownerinfo.owncar.socsecurity, а если значение flits.status равно 1 – то flits.ownerinfo.leasecar.name.

Анонимные объединения (C11)

Анонимные объединения работают во многом подобно анонимным структурам. То есть анонимное объединение – это неименованное объединение, являющееся членом структуры или объединения. Например, структуру car_data можно переопределить следующим образом:

```
struct owner {
   char socsecurity[12];
   ...
};
struct leasecompany {
   char name[40];
   char headquarters[40];
   ...
};
struct car_data {
   char make[15];
   int status;  /* 0 = принадлежит, 1 = взят напрокат */
   union {
      struct owner owncar;
      struct leasecompany leasecar;
   };
   ...
};
```

Теперь, если flits – это структура car_data, мы можем применять flits.owncar.socsecurity вместо flits.ownerinfo.owncar.socsecurity.

Сводка: операции со структурами и объединениями

Операция членства: .

Общий комментарий

Эта операция используется с именем структуры или объединения для указания члена структуры или объединения. Если name — имя структуры, a member — член, описанный шаблоном структуры, то следующее выражение идентифицирует этот член структуры:

```
name.member
```

Типом члена name.member является тип, указанный для member. Операция членства также может применяться и с объединениями.

Пример

```
struct {
   int code;
   float cost;
} item;
item.code = 1265;
```

Последний оператор присваивает значение члену code структуры item.

Операция косвенного членства: ->

Общий комментарий

Эта операция используется с указателем на структуру или объединение с целью идентификации члена структуры или объединения. Предположим, что ptrstr является указателем на структуру, а member — членом, описанным в шаблоне структуры. Тогда оператор

```
ptrstr->member
```

идентифицирует этот член указанной структуры. В аналогичной манере операция косвенного членства может применяться с объединениями.

Пример

```
struct {
   int code;
   float cost;
} item, * ptrst;

ptrst = &item;
ptrst->code = 3451;
```

Последний оператор присваивает значение int члену code структуры item. Три приведенных ниже выражения эквивалентны:

```
ptrst->code    item.code    (*ptrst).code
```

Перечислимые типы

Перечислимый тип можно использовать для объявления символических имен, представляющих целочисленные константы. Ключевое слово enum позволяет создать новый "тип" и указать значения, которые для него допускаются. (На самом деле константы enum имеют тип int, поэтому их можно применять везде, где разрешено использовать тип int.) Целью перечислимых типов является улучшение читабельности программы. Их синтаксис похож на синтаксис, применяемый для структур. Например, можно записать следующие объявления:

```
enum spectrum {red, orange, yellow, green, blue, violet};
enum spectrum color;
```

Первое объявление устанавливает spectrum как имя дескриптора, который позволяет использовать enum spectrum в качестве имени типа. Второе объявление делает color переменной этого типа. Идентификаторы внутри фигурных скобок перечисляют возможные значения, которые может иметь переменная spectrum. Таким образом, возможными значениями color будут red, orange, yellow и т.д. Эти символические константы называются *перечислителями*. Затем допускается применение показанных ниже операторов:

```
int c;
color = blue;
if (color == yellow)
   ...;
for (color = red; color <= violet; color++)
   ...;
```

Хотя перечислители вроде red и blue имеют тип int, переменные перечислимого типа не так жестко привязаны к целочисленному типу до тех пор, пока этот тип может содержать перечислимые константы. Например, перечислимые константы для spectrum входят в диапазон 0–5, так что для представления переменной color компилятор мог бы выбрать тип unsigned char.

Кстати, некоторые свойства перечислений С не переносятся в С++. Например, С позволяет применять к перечислимой переменной операцию ++, но стандарт С++ этого не допускает. Таким образом, если вы предполагаете, что в будущем код может быть объединен с программой С++, то должны объявить переменную color в предыдущем примере как относящуюся к типу int. Тогда код будет работать как в С, так и в С++.

Константы enum

Так что собой представляют blue и red? Формально они являются константами типа int. Например, имея предыдущее объявление перечислимого типа, можно записать так:

```
printf("red = %d, orange = %d\n", red, orange);
```

Ниже показан вывод:

```
red = 0, orange = 1
```

Оказалось, что red стала именованной константой, представляющей целочисленное значение 0. Подобным же образом другие идентификаторы являются именованными константами, представляющими целые числа от 1 до 5. Перечислимую константу можно использовать везде, где допускается применение целочисленной константы. Например, их можно использовать для указания размеров в объявлениях массивов или в качестве меток в операторе switch.

Стандартные значения

По умолчанию константам в списке перечислений присваиваются целочисленные значения 0, 1, 2 и т.д. Следовательно, объявление

```
enum kids {nippy, slats, skippy, nina, liz};
```

приводит к тому, что nina имеет значение 3.

Присвоенные значения

При желании вы можете выбрать целочисленные значения, которые должны иметь константы. Для этого просто включите нужные значения в объявление:

```
enum levels {low = 100, medium = 500, high = 2000};
```

Если значение присваивается одной константе, но не следующим за ней, то дальнейшие константы получат последовательно возрастающие значения. Например, взгляните на следующее объявление:

```
enum feline {cat, lynx = 10, puma, tiger};
```

В этом случае cat получает стандартное значение 0, а lynx, puma и tiger – соответственно, 10, 11 и 12.

Использование enum

Вспомните, что целью перечислимых типов является улучшение читабельности программы и упрощение ее сопровождения. Если вы имеете дело с цветами, то применение red (красный) и blue (голубой) намного информативнее, чем указание значений 0 и 1. Обратите внимание, что перечислимые типы предназначены для внутреннего использования. Если вы хотите ввести значение orange для переменной color, то должны вводить 1, а не слово orange, или же можно прочитать строку "orange" и заставить программу преобразовать ее в значение orange.

Из-за того, что перечислимый тип является целочисленным, переменные enum могут применяться в выражениях таким же образом, как целочисленные переменные. Они представляют собой удобные метки для операторов case.

В листинге 14.15 приведен краткий пример использования enum. Пример полагается на стандартную схему присваивания значений. В результате константа red получает значение 0, которое делает ее индексом для указателя на строку "red".

Листинг 14.15. Программа enum.c

```
/* enum.c -- использование перечислимых значений */
#include <stdio.h>
#include <string.h>   // для strcmp(), strchr()
#include <stdbool.h>  // средство C99
char * s_gets(char * st, int n);
enum spectrum {red, orange, yellow, green, blue, violet};
const char * colors[] = {"red", "orange", "yellow",
   "green", "blue", "violet"};
#define LEN 30

int main(void)
{
   char choice[LEN];
   enum spectrum color;
   bool color_is_found = false;

   puts("Введите цвет (или пустую строку для выхода):");
   while (s_gets(choice, LEN) != NULL && choice[0] != '\0')
   {
      for (color = red; color <= violet; color++)
      {
         if (strcmp(choice, colors[color]) == 0)
         {
            color_is_found = true;
            break;
         }
      }
      if (color_is_found)
      switch(color)
      {
      case red : puts("Розы красные.");
         break;
      case orange : puts("Маки оранжевые.");
         break;
      case yellow : puts("Подсолнухи желтые.");
         break;
      case green : puts("Трава зеленая.");
         break;
      case blue : puts("Колокольчики синие.");
         break;
      case violet : puts("Фиалки фиолетовые.");
         break;
      }
      else
         printf("Цвет %s не известен.\n", choice);
      color_is_found = false;
      puts("Введите следующий цвет (или пустую строку для выхода):");
   }
   puts("Программа завершена.");
   return 0;
}
```

```
char * s_gets(char * st, int n)
{
   char * ret_val;
   char * find;
   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      find = strchr(st, '\n');     // поиск новой строки
      if (find)                    // если адрес не равен NULL,
         *find = '\0';             // поместить туда нулевой символ
      else
         while (getchar() != '\n')
            continue;              // отбросить остаток строки
   }
   return ret_val;
}
```

Цикл for завершается, когда входная строка совпадает с одной из строк, на которые указывают элементы массива colors. Если цикл находит совпадающий цвет, то значение перечислимой переменной применяется для сопоставления с перечислимой константой, используемой в качестве метки case. Ниже приведены результаты пробного запуска:

```
Введите цвет (или пустую строку для выхода):
blue
Колокольчики синие.
Введите следующий цвет (или пустую строку для выхода):
orange
Маки оранжевые.
Введите следующий цвет (или пустую строку для выхода):
purple
Цвет purple не известен.
Введите следующий цвет (или пустую строку для выхода):
Программа завершена.
```

Совместно используемые пространства имен

Термин *пространство имен* в языке С применяется для идентификации частей программы, в которых распознается то или иное имя. Область видимости входит в состав этой концепции: две переменные, имеющие одно и то же имя, но разные области видимости, не конфликтуют друг с другом, в отличие от двух переменных с одинаковыми именами и одной и той же областью видимости. Существует также аспект категории, относящийся к пространствам имен. Дескрипторы структур, дескрипторы объединений и дескрипторы перечислений в определенной области видимости совместно используют одно и то же пространство имен, и это пространство имен отличается от пространства, применяемого обычными переменными. Это означает, что можно назначить одинаковые имена переменной и дескриптору в рамках одной и той же области видимости без возникновения ошибки, но нельзя объявлять два дескриптора или две переменных с одним и тем же именем в той же самой области видимости. Например, следующие объявления не приводят к конфликту имен в С:

```
struct rect { double x; double y; };
int rect;  // конфликт в С не возникает
```

Тем не менее, использование одного идентификатора двумя разными путями может вызвать путаницу, к тому же это не разрешено в С++, т.к. там дескрипторы и переменные помещаются в то же самое пространство имен.

Средство typedef: краткое знакомство

Возможность typedef представляет собой усовершенствованное средство манипулирования данными, которое позволяет создавать собственное имя для типа. В этом отношении оно подобно директиве #define, но с тремя отличиями.

- В отличие от #define, средство typedef ограничено назначением символических имен только типам, но не значениям.
- Интерпретация typedef выполняется компилятором, а не препроцессором.
- В рамках своих ограничений средство typedef является более гибким, чем #define.

Давайте посмотрим, как работает typedef. Предположим, что вы хотите использовать элемент BYTE для обозначения однобайтовых чисел. Тогда вы просто объявляете BYTE, как если бы это была переменная типа char, и предваряете определение ключевым словом typedef:

```
typedef unsigned char BYTE;
```

После этого BYTE можно применять для определения переменных:

```
BYTE x, y[10], * z;
```

Область видимости этого определения зависит от местоположения оператора typedef. Если определение находится внутри функции, область видимости будет локальной в пределах этой функции. Если определение находится вне функции, область видимости будет глобальной.

Для определений typedef часто используются прописные буквы, чтобы напоминать пользователю о том, что имя типа в действительности является символическим сокращением, но разрешены также и строчные буквы:

```
typedef unsigned char byte;
```

Имена в typedef подчиняются тем же правилам, которые регламентируют создание допустимых имен переменных.

Хотя создание имени для существующего типа может показаться незначительной возможностью, часто оно будет удобным. В предыдущем примере указание BYTE вместо unsigned char помогает документировать намерение применять переменные BYTE для представления чисел, а не символьных кодов. Использование typedef также способствует лучшей переносимости. Например, ранее мы упоминали о типе size_t, который представляет тип, возвращаемый операцией sizeof, и о типе time_t, представляющем тип, который возвращается функцией time(). Стандарт С утверждает, что sizeof и time() возвращают целочисленные типы, но то, какими они должны быть, оставляет на усмотрение реализации. Причина отсутствия конкретики объясняется тем, что в комитете по стандартам С придерживаются мнения, что, скорее всего, не существует единого выбора, который был бы наилучшим для всех компьютерных платформ. Таким образом, было решено создать новое имя типа, подобное time_t, и позволить реализациям применять typedef для установки этого имени в какой-то конкретный тип. Тогда появляется возможность предоставить общий прототип, такой как показанный ниже:

```
time_t time(time_t *);
```

В одной системе time_t может быть unsigned long, а в другой — unsigned long long. При условии, что включен заголовочный файл time.h, программа может получать доступ к подходящему определению, и в коде можно объявлять переменные типа time_t.

Некоторые возможности typedef можно продублировать с помощью #define. Например, указание

```
#define BYTE unsigned char
```

заставляет препроцессор заменять BYTE типом unsigned char. Ниже приведен пример typedef, который невозможно воспроизвести посредством #define:

```
typedef char * STRING;
```

Без ключевого слова typedef в этом примере сама переменная STRING идентифицировалась бы как указатель на char. Наличие typedef делает STRING идентификатором для указателей на char. Таким образом,

```
STRING name, sign;
```

означает

```
char * name, * sign;
```

Предположим, что вместо этого вы поступили так:

```
#define STRING char *
```

Тогда

```
STRING name, sign;
```

транслируется в

```
char * name, sign;
```

В этом случае указателем будет только name.

Средство typedef можно также использовать со структурами:

```
typedef struct complex {
   float real;
   float imag;
} COMPLEX;
```

Теперь для представления комплексных чисел вместо структуры по имени complex можно применять тип COMPLEX. Одна из целей использования typedef связана с созданием удобных и легко опознаваемых имен для часто встречающихся типов. Например, многие программисты предпочитают применять имя типа STRING или его эквивалент, как в ранее показанном примере. При использовании typedef для именования типа структуры дескриптор можно не указывать:

```
typedef struct {double x; double y;} rect;
```

Предположим, что определенный посредством typedef идентификатор применяется так, как показано ниже:

```
rect r1 = {3.0, 6.0};
rect r2;
```

Этот код транслируется в следующие операторы:

```
struct {double x; double y;} r1 = {3.0, 6.0};
struct {double x; double y;} r2;
r2 = r1;
```

Если две структуры объявлены без дескриптора, но с идентичными членами (совпадают имена членов и типов), то в C эти две структуры считаются имеющими один и тот же тип, поэтому присваивание r1 переменной r2 является допустимой операцией.

Вторая причина использования typedef связана с тем, что имена typedef часто применяются для сложных типов. Например, объявление

```
typedef char (* FRPTC ()) [5];
```

делает FRPTC идентификатором типа, который является функцией, возвращающей указатель на массив из 5 элементов char. (В следующем разделе обсуждаются причудливые объявления.)

При использовании средства typedef имейте в виду, что оно не создает новые типы; вместо этого вы получаете всего лишь удобные метки. Это значит, например, что переменные, применяющие созданный нами тип STRING, могут передаваться в качестве аргументов функциям, которые ожидают тип указателя на char.

Благодаря структурам, объединениям и typedef, язык C предоставляет инструменты для эффективной и переносимой обработки данных.

Причудливые объявления

Язык C позволяет создавать сложные формы данных. Хотя мы придерживаемся простейших форм, все же имеет смысл отметить некоторые из доступных возможностей. Когда вы делаете объявление, имя (или идентификатор) можно изменить, добавив модификатор.

Модификатор	Описание
*	Обозначает указатель
()	Обозначает функцию
[]	Обозначает массив

Одновременно в C разрешено указывать несколько модификаторов, что, в свою очередь, позволяет создавать широкое разнообразие типов, как продемонстрировано в следующих примерах:

```
int board[8][8];      // массив из массивов значений int
int ** ptr;           // указатель на указатель на int
int * risks[10];      // 10-элементный массив указателей на int
int (* rusks)[10];    // указатель на массив из 10 значений int
int * oof[3][4];      // массив размером 3 x 4 указателей на int
int (* uuf)[3][4];    // указатель на массив размером 3 x 4 значений int
int (* uof[3])[4];    // 3-элементный массив указателей на 4-элементные
                      // массивы значений int
```

Трюк по распутыванию таких объявлений заключается в определении порядка, в котором необходимо использовать модификаторы. Описанные ниже правила должны дать вам необходимое представление об этом.

1. Скобки [], которые обозначают массив, и скобки (), обозначающие функцию, имеют одинаковый приоритет. Этот приоритет выше, чем у операции разыменования *, которая означает, что следующее объявление делает risks массивом указателей, а не указателем на массив:

   ```
   int * risks[10];
   ```

2. Скобки [] и () имеют ассоциативность слева направо. Таким образом, приведенное ниже объявление делает goods массивом из 12 массивов, содержащих по 50 значений int, а не массивом из 50 массивов с 12 элементами типа int:

   ```
   int goods[12][50];
   ```

3. Скобки [] и () имеют один и тот же приоритет, но из-за их ассоциативности слева направо в следующем объявлении * и rusks группируются вместе перед применением квадратных скобок. Это означает, что объявление делает rusks указателем на массив из 10 значений int:

```
int (* rusks)[10];
```

Давайте воспользуемся этими правилами для разбора показанного далее объявления:

```
int * oof[3][4];
```

Конструкция [3] имеет более высокий приоритет, чем *, и поскольку действует правило ассоциативности слева направо, она применяется перед [4]. Таким образом, oof — это массив из трех элементов. Следующим по порядку идет [4], поэтому элементами oof являются массивы из четырех элементов. Модификатор * сообщает о том, что эти элементы представляют собой указатели. Картину завершает int: итак, oof представляет собой трехэлементный массив, состоящий из четырехэлементных массивов указателей на int, или для краткости массив 3×4 указателей на int. Память выделяется под 12 указателей.

Теперь взгляните на следующее объявление:

```
int (* uuf)[3][4];
```

Круглые скобки приводят к тому, что модификатор * получает первый приоритет, благодаря чему uuf становится указателем на массив 3×4 значений int. Память выделяется только для одного указателя.

Эти правила также позволяют иметь такие типы:

```
char * fump(int);        // функция, возвращающая указатель на char
char (* frump)(int);     // указатель на функцию, возвращающую тип char
char (* flump[3])(int);  // массив из 3 указателей на функции, которые
                         // возвращают тип char
```

Все три функции принимают аргумент int.

Средство typedef можно использовать для построения последовательности связанных друг с другом типов данных:

```
typedef int arr5[5];
typedef arr5 * p_arr5;
typedef p_arr5 arrp10[10];
arr5 togs; // togs - массив из 5 значений int
p_arr5 p2; // p2 - указатель на массив из 5 значений int
arrp10 ap; // ap – массив, содержащий 10 указателей на массивы из 5 значений int
```

Когда вы освоитесь с созданием структур подобного рода, возможности для самых причудливых объявлений действительно возрастут. Что же касается их применений, то мы оставляем эти вопросы за более сложными источниками.

Функции и указатели

Как продемонстрировало обсуждение объявлений, допускается объявлять указатели на функции. Возможно, вас интересует, в чем они могут быть полезны. Обычно указатель на функцию используется в качестве аргумента в другой функции, сообщая ей, какую функцию применять. Например, сортировка массива предполагает сравнение двух элементов для выяснения того, какой из них должен следовать первым. В случае числовых элементов можно использовать операцию >. Но в целом элементом может быть строка или структура, что требует вызова специальной функции для выполнения

сравнения. Функция qsort() из библиотеки C спроектирована на работу с массивами любого вида при условии, что вы уведомите ее о том, какую функцию применять для сравнения элементов. С этой целью qsort() принимает в одном из своих аргументов указатель на функцию. Затем qsort() использует указанную функцию для сортировки значений определенного типа — будь он целочисленным, строкой или структурой.

Давайте подробнее рассмотрим указатели на функции. Прежде всего, что они означают? Скажем, указатель на int содержит адрес ячейки памяти, в которой может быть сохранено значение int. Функции также имеют адреса, поскольку реализация функции на машинном языке состоит из кода, загружаемого в память. Указатель на функцию может содержать адрес, помечающий начало кода функции.

Далее, когда вы объявляете указатель на данные, то должны объявить тип данных, на которые он указывает. При объявлении указателя на функцию необходимо объявить тип указываемой функции. Чтобы задать тип функции, понадобится указать сигнатуру функции, т.е. возвращаемый тип и типы параметров функции. Например, взгляните на следующий прототип:

```
void ToUpper(char *); // преобразует строку в верхний регистр
```

Тип ToUpper() определен как "функция с параметром char * и возвращаемым типом void". Вот как объявить указатель на функцию такого типа по имени pf:

```
void (*pf)(char *);  // pf - указатель на функцию
```

Читая это объявление, вы видите, что первая пара круглых скобок связывает операцию * с pf, т.е. pf является указателем на функцию. Это делает (*pf) функцией, а (char *) — списком ее параметров функции и void — возвращаемым типом. Вероятно, проще всего понять, как создано такое объявление — обратить внимание, что имя функции ToUpper в нем заменено выражением (*pf). Таким образом, если вы хотите объявить указатель на специфичный тип функции, можете объявить функцию этого типа и затем заменить имя функции выражением вида (*pf), получив в результате объявление указателя на функцию. Как упоминалось ранее, первые круглые скобки необходимы из-за правил, регламентирующих приоритеты операций. Если их отбросить, получится что-то совершенно другое:

```
void *pf(char *);    // pf - функция, которая возвращает указатель
```

Совет

Чтобы объявить указатель на функцию конкретного типа, сначала объявите функцию желаемого типа и затем замените имя функции выражением в форме (*pf); после этого pf становится указателем на функцию данного типа.

Указателю на функцию можно присваивать адрес функции подходящего типа. В этом контексте для представления адреса функции может применяться ее *имя*:

```
void ToUpper(char *);
void ToLower(char *);
int round(double);
void (*pf)(char *);
pf = ToUpper;        // допустимо, ToUpper - адрес функции
pf = ToLower;        // допустимо, ToLower - адрес функции
pf = round;          // недопустимо, round - неподходящий тип функции
pf = ToLower();      // недопустимо, ToLower() не является адресом
```

Последнее присваивание также недопустимо, потому что нельзя использовать функцию void в операторе присваивания.

Обратите внимание, что указатель pf может указывать на любую функцию, которая принимает аргумент char * и имеет возвращаемый тип void, но не на функции с другими характеристиками.

Подобно тому, как можно применять указатель на данные с целью доступа к ним, вы можете использовать указатель на функцию для обращения к этой функции. На удивление для этого существуют два логически несогласованных синтаксических правила, как иллюстрируется в следующем фрагменте:

```
void ToUpper(char *);
void ToLower(char *);
void (*pf)(char *);
char mis[] = "Nina Metier";
pf = ToUpper;
(*pf)(mis);        // применить ToUpper к mis (первый синтаксис)
pf = ToLower;
pf(mis);           // применить ToLower к mis (второй синтаксис)
```

Каждый подход выглядит логичным. Проанализируем первый подход: так как pf указывает на функцию ToUpper, то *pf — это функция ToUpper, поэтому выражение (*pf)(mis) аналогично ToUpper(mis). Чтобы убедиться в эквивалентности ToUpper и (*pf), достаточно взглянуть на объявления ToUpper и pf. Второй подход можно объяснить так: из-за того, что имя функции является указателем, указатель и имя функций можно применять взаимозаменяемо, следовательно, pf(mis) — это то же самое, что и ToLower(mis). Чтобы удостовериться в эквивалентности pf и ToLower, просто посмотрите на оператор присваивания для pf. Исторически сложилось так, что разработчики C и Unix в Bell Labs избрали первый подход, а разработчики, которые расширяли Unix в Беркли, приняли второй подход. Компилятор K&R C не разрешает вторую форму, но для поддержки совместимости с существующим кодом стандарт ANSI C принимает обе формы ((*pf)(mis) и pf(mis)) как эквивалентные. Последующие стандарты сохранили такой в высшей степени двойственный подход.

Одним из наиболее распространенных случаев использования указателей на данные является аргумент функции, и то же самое относится к указателю на функцию. Например, рассмотрим следующий прототип функции:

```
void show(void (* fp)(char *), char * str);
```

Он выглядит запутанным, но в нем объявляются два параметра, fp и str. Параметр fp — это указатель на функцию, а str — указатель на данные. Точнее, fp указывает на функцию, которая принимает параметр char * и имеет возвращаемый тип void, а str указывает на char. Таким образом, имея представленные выше объявления, можно делать вызовы функций вроде приведенных ниже:

```
show(ToLower, mis); /* show() использует функцию ToLower(): fp = ToLower   */
show(pf, mis); /*show() использует функцию, указанную посредством pf: fp = pf */
```

Каким образом show() применяет переданный указатель на функцию? Для вызова этой функции в show() используется либо синтаксис fp(), либо синтаксис (*fp)():

```
void show(void (* fp)(char *), char * str)
{
   (*fp)(str); /* применить выбранную функцию к str */
   puts(str);  /* отобразить результат              */
}
```

Здесь функция show() сначала трансформирует строку str, применяя к ней функцию, на которую указывает fp, после чего отображает результирующую строку.

Кстати говоря, функции с возвращаемыми значениями могут использоваться двумя разными способами при передаче в качестве аргументов другим функциям. Например, взгляните на следующие операторы:

```
function1(sqrt);        /* передает адрес функции sqrt              */
function2(sqrt(4.0));   /* передает возвращаемое значение функции sqrt */
```

Первый оператор передает адрес функции `sqrt()`, и предположительно `function1()` будет применять эту функцию в своем коде. Второй оператор сначала вызывает функцию `sqrt()` и затем передает возвращаемое значение (в этом случае 2.0) функции `function2()`.

Для демонстрации основных идей в листинге 14.16 используется функция `show()` вместе с набором функций трансформации в качестве аргументов. В листинге также продемонстрировано несколько полезных приемов поддержки меню.

Листинг 14.16. Программа `func_ptr.c`

```
// func_ptr.c -- использование указателей на функции
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define LEN 81
char * s_gets(char * st, int n);
char showmenu(void);
void eatline(void);          // читает до конца строки
void show(void (* fp)(char *), char * str);
void ToUpper(char *);        // преобразует строку в верхний регистр
void ToLower(char *);        // преобразует строку в нижний регистр
void Transpose(char *);      // меняет местами регистры
void Dummy(char *);          // оставляет строку неизмененной

int main(void)
{
   char line[LEN];
   char copy[LEN];
   char choice;
   void (*pfun)(char *);     // указывает на функцию, которая имеет аргумент
                             // типа char * и ничего не возвращает
   puts("Введите строку (пустая строка - выход из программы):");
   while (s_gets(line, LEN) != NULL && line[0] != '\0')
   {
      while ((choice = showmenu()) != 'n')
      {
         switch (choice )    // оператор switch устанавливает указатель
         {
            case 'u' : pfun = ToUpper; break;
            case 'l' : pfun = ToLower; break;
            case 't' : pfun = Transpose; break;
            case 'o' : pfun = Dummy; break;
         }
         strcpy(copy, line); // сделать копию для show()
         show(pfun, copy);   // использовать выбранную функцию
      }
      puts("Введите строку (пустая строка - выход из программы):");
   }
   puts("Программа завершена.");

   return 0;
}
```

```
char showmenu(void)
{
   char ans;
   puts("Введите выбранный вариант из меню:");
   puts("u) нижний регистр             l) верхний регистр");
   puts("t) поменять местами регистры  o) исходный регистр");
   puts("n) следующая строка");
   ans = getchar();       // получить ответ
   ans = tolower(ans);    // преобразовать в нижний регистр
   eatline();             // избавиться от оставшейся части строки
   while (strchr("ulton", ans) == NULL)
   {
      puts("Введите u, l, t, o или n:");
      ans = tolower(getchar());
      eatline();
   }
   return ans;
}
void eatline(void)
{
   while (getchar() != '\n')
   continue;
}
void ToUpper(char * str)
{
   while (*str)
   {
      *str = toupper(*str);
      str++;
   }
}
void ToLower(char * str)
{
   while (*str)
   {
      *str = tolower(*str);
      str++;
   }
}
void Transpose(char * str)
{
   while (*str)
   {
      if (islower(*str))
         *str = toupper(*str);
      else if (isupper(*str))
         *str = tolower(*str);
      str++;
   }
}
void Dummy(char * str)
{
   // оставляет строку неизмененной
}
```

```
void show(void (* fp)(char *), char * str)
{
   (*fp)(str); // применить выбранную функцию к str
   puts(str);  // отобразить результат
}
char * s_gets(char * st, int n)
{
   char * ret_val;
   char * find;
   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      find = strchr(st, '\n');     // поиск новой строки
      if (find)                    // если адрес не равен NULL,
         *find = '\0';             // поместить туда нулевой символ
      else
         while (getchar() != '\n')
            continue;              // отбросить остаток строки
   }
   return ret_val;
}
```

Ниже показаны результаты пробного запуска:

```
Введите строку (пустая строка - выход из программы):
Does C make you feel loopy?
Введите выбранный вариант из меню:
u) нижний регистр            l) верхний регистр
t) поменять местами регистры  o) исходный регистр
n) следующая строка
t
dOES c MAKE YOU FEEL LOOPY?
Введите выбранный вариант из меню:
u) нижний регистр            l) верхний регистр
t) поменять местами регистры  o) исходный регистр
n) следующая строка
l
does c make you feel loopy?
Введите выбранный вариант из меню:
u) нижний регистр           l) верхний регистр
t) поменять местами регистры  o) исходный регистр
n) следующая строка
n
Введите строку (пустая строка - выход из программы):
Программа завершена.
```

Обратите внимание, что функции ToUpper(), ToLower(), Transpose() и Dummy() имеют тот же самый тип, поэтому все они могут быть присвоены указателю pfun. В этой программе в качестве аргумента для show() применяется pfun, но можно также указывать непосредственно любое из имен четырех функций, как в show(Transpose, copy).

В ситуациях подобного рода можно использовать typedef. Например, в программе можно было бы предусмотреть следующие операторы:

```
typedef void (*V_FP_CHARP)(char *);
void show (V_FP_CHARP fp, char *);
V_FP_CHARP pfun;
```

Если вы склонны к приключениям, можете объявить и инициализировать массив таких указателей:

```
V_FP_CHARP arpf[4] = {ToUpper, ToLower, Transpose, Dummy};
```

Если затем модифицировать функцию showmenu() так, чтобы она превратилась в тип int и возвращала значение 0, когда пользователь вводит символ u, значение 1 — когда l, значение 2 — когда t и т.д., то цикл, содержащий switch, можно заменить приведенным ниже кодом:

```
index = showmenu();
while (index >= 0 && index <= 3)
{
    strcpy(copy, line);       /* сделать копию для show()       */
    show(arpf[index], copy);  /* использовать выбранную функцию */
    index = showmenu();
}
```

Нельзя иметь массив функций, но можно иметь массив указателей на функции.

К этому моменту вы ознакомились со всеми четырьмя способами применения имени функции: в определении функции, в объявлении функции, в вызове функции и в качестве указателя (рис. 14.4).

Имя функции, используемое в объявлении прототипа:	`int comp(int x, int y);`
Имя функции, используемое в вызове функции:	`status = comp(q,r);`
Имя функции, используемое в определении функции:	`int comp(intx, inty)` `{ ...`
Имя функции, используемое в качестве указателя при присваивании:	`pfunct = comp;`
Имя функции, используемое в качестве аргумента типа указателя:	`slowsort(arr,n,comp);`

Рис. 14.4. Использование имени функции

В плане поддержки меню функция showmenu() демонстрирует несколько приемов. Прежде всего, код

```
ans = getchar();      // получить ответ
ans = tolower(ans);   // преобразовать в нижний регистр
```

и

```
ans = tolower(getchar());
```

отражает два способа преобразования пользовательского ввода в один регистр, так что не приходится выполнять проверку для 'u' и 'U' и т.д.

Функция eatline() избавляется от оставшейся части введенной строки. Это полезно по двум причинам. Во-первых, при выборе действия в меню пользователь набирает букву, а затем нажимает клавишу <Enter>, что приводит к генерации символа новой строки. Если первым делом не избавиться от этого символа, он будет прочитан как следующий ответ. Во-вторых, предположим, что вместо буквы *u* пользователь вводит слово *uppercase* целиком. Без функции eatline() программа трактовала бы каждый символ слова *uppercase* как отдельный ответ. Благодаря eatline(), программа обрабатывает символ *u* и отбрасывает оставшуюся часть строки.

Далее, функция `showmenu()` спроектирована так, чтобы возвращать в программу только допустимые варианты выбора. Для содействия в этом применяется стандартная библиотечная функция `strchr()` из заголовочного файла `string.h`:

```
while (strchr("ulton", ans) == NULL)
```

Эта функция ищет местоположение первого вхождения символа `ans` в строке `"ulton"` и возвращает указатель на него. Если символ не найден, возвращается нулевой указатель. Таким образом, использованная нами в цикле `while` проверка удобнее конструкции следующего вида:

```
while (ans != 'u' && ans != 'l' && ans != 't' && ans != 'o' && ans != 'n')
```

Чем больше вариантов приходится проверять, тем более удобным становится применение функции `strchr()`.

Ключевые понятия

Информация, необходимая при решении задачи по программированию, часто выходит за рамки одиночного числа или списка чисел. Программа может иметь дело с сущностью или коллекцией сущностей, обладающих множеством свойств. Например, заказчик может быть представлен с помощью его имени, фамилии, адреса, номера телефона и других сведений; DVD-диск с фильмом может быть описан посредством его названия, продавца, длительности фильма, стоимости и прочих данных. Структура C позволяет собирать всю информацию вместе в одном элементе. Это очень удобно для организации программы. Вместо того чтобы хранить информацию в разрозненных переменных, все связанные данные сохраняются в одном месте.

При проектировании структуры часто полезно разработать пакет функций и в дальнейшем иметь дело только с ним. Например, вместо написания множества операторов `printf()` каждый раз, когда требуется отобразить содержимое структуры, можно написать функцию отображения, которая принимает структуру (или ее адрес) в качестве аргумента. Поскольку вся информация находится в структуре, для функции достаточно только одного аргумента. Если бы данные были помещены в отдельные переменные, для каждой порции данных пришлось бы предусмотреть свой аргумент. Кроме того, если вы, скажем, добавите в структуру еще один член, то нужно будет переписать функции, но не их вызовы, что является большим удобством во время модификации проектного решения.

Объявление объединения во многом похоже на объявление структуры. Однако члены объединения совместно используют одно и то же пространство памяти, и в каждый конкретный момент времени только один член может содержать данные. По существу объединение позволяет создать переменную, которая может содержать одно значение, но разных типов.

Средство `enum` предлагает инструмент для определения символических констант, а `typedef` – средство создания нового идентификатора для базового или производного типа.

Указатели на функции предоставляют инструмент для уведомления одной функции о том, какие функции она должна применять.

Резюме

Структуры C служат средством для хранения нескольких элементов данных, обычно разных типов, в одном и том же объекте данных. Для идентификации шаблона структуры и для объявления переменных этого типа можно использовать дескриптор. Операция членства (.) позволяет обращаться к индивидуальным членам структуры с применением меток из шаблона структуры.

Имея указатель на структуру, для доступа к отдельным ее членам можно использовать этот указатель и операцию косвенного членства (->) вместо имени и операции точки. Получить адрес структуры можно с помощью операции &. В отличие от массивов, имя структуры не служит ее адресом.

Традиционно функции, связанные со структурами, принимали в своих аргументах указатели на структуры. Современная версия языка C допускает передачу структур в качестве аргументов, применение структур как возвращаемых значений и присваивание структур одного и того же типа. Тем не менее, передача адреса обычно эффективнее.

Объединения имеют тот же самый синтаксис, что и структуры. Однако члены в объединениях совместно используют одно и то же пространство памяти. Вместо того чтобы одновременно хранить несколько типов данных, как это делает структура, объединение хранит элемент данных одного типа из списка вариантов. Другими словами, структура может хранить, скажем, данные `int`, `double` и `char`, а соответствующее объединение может содержать либо `int`, либо `double`, либо `char`.

Перечисления позволяют создавать группу символических целочисленных констант (перечислимых констант) и определять связанный перечислимый тип.

С помощью средства `typedef` можно устанавливать псевдонимы или сокращенные представления стандартных типов C.

Имя функции является ее адресом. Такие адреса могут передаваться в виде аргументов функциям, которые в дальнейшем применяют функции, указанные этими адресами. Если `pf` — указатель на функцию, которому был присвоен адрес конкретной функции, то эту функцию можно вызвать двумя способами:

```
#include <math.h>    /* объявление функции double sin(double) */
...
double (*pdf)(double);
double x;
pdf = sin;
x = (*pdf)(1.2);      // вызывает sin(1.2)
x = pdf(1.2);         // также вызывает sin(1.2)
```

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля приведены в приложении А.

1. Что неправильно в следующем шаблоне?

```
structure {
      char itable;
      int num[20];
      char * togs
}
```

2. Ниже приведен фрагмент программы. Что он выведет?

```
#include <stdio.h>
struct house {
   float sqft;
```

```
    int rooms;
    int stories;
    char address[40];
};
int main(void)
{
    struct house fruzt = {1560.0, 6, 1, "22 Spiffo Road"};
    struct house *sign;
    sign = &fruzt;
    printf("%d %d\n", fruzt.rooms, sign->stories);
    printf("%s \n", fruzt.address);
    printf("%c %c\n", sign->address[3], fruzt.address[4]);
    return 0;
}
```

3. Разработайте шаблон структуры, которая будет содержать название месяца, его трехбуквенную аббревиатуру, количество дней в месяце и его номер.

4. Определите массив из 12 структур вида, описанного в вопросе 3, и инициализируйте ее для года, который не является високосным.

5. Напишите функцию, которая после получения номера месяца возвращает общее количество дней с начала года до конца указанного месяца. Предположите, что шаблон структуры, описанный в вопросе 3, и подходящий массив таких структур объявлены внешне.

6. а. Имея показанное ниже определение `typedef`, объявите 10-элементный массив указанных структур. Затем с помощью присваивания индивидуальных членов сделайте так, чтобы третий элемент этого массива описывал объектив марки Remarkatar с фокусным расстоянием 500 мм и диафрагмой f/2.0.

```
typedef struct lens {     /* дескриптор структуры lens          */
    float foclen;         /* фокусное расстояние в миллиметрах */
    float fstop;          /* диафрагма                          */
    char brand[30];       /* марка производителя                */
} LENS;
```

   б. Повторите часть а), но воспользуйтесь в объявлении списком инициализации с назначенным инициализатором вместо написания отдельных операторов присваивания для каждого члена.

7. Взгляните на следующий фрагмент кода:

```
struct name {
        char first[20];
        char last[20];
};
struct bem {
        int limbs;
        struct name title;
        char type[30];
};
struct bem * pb;
struct bem deb = {
        6,
        {"Berbnazel", "Gwolkapwolk"},
        "Arcturan"
};
pb = &deb;
```

а. Что выведут приведенные ниже операторы?

```
printf("%d\n", deb.limbs);
printf("%s\n", pb->type);
printf("%s\n", pb->type + 2);
```

б. Как можно было бы представить "Gwolkapwolk" в форме записи для структур (двумя способами)?

в. Напишите функцию, которая принимает адрес структуры bem в качестве своего аргумента и выводит содержимое этой структуры в показанной ниже форме (предположите, что шаблон структуры находится в заголовочном файле starfolk.h):

```
Berbnazel Gwolkapwolk - это 6-конечный Arcturan.
```

8. Взгляните на следующие объявления:

```
struct fullname {
            char fname[20];
            char lname[20];
            };
struct bard   {
            struct fullname name;
            int born;
            int died;
            };
struct bard willie;
struct bard *pt = &willie;
```

а. Идентифицируйте член born структуры willie с помощью идентификатора willie.

б. Идентифицируйте член born структуры willie с помощью идентификатора pt.

в. С помощью функции scanf() прочитайте значение для члена born, используя идентификатор willie.

г. С помощью функции scanf() прочитайте значение для члена born, используя идентификатор pt.

д. С помощью функции scanf() прочитайте значение для члена lname структуры name, используя идентификатор willie.

е. С помощью функции scanf() прочитайте значение для члена lname структуры name, используя идентификатор pt.

ж. Создайте идентификатор для третьей буквы фамилии того, кто описан переменной willie.

з. Напишите выражение, которое представляет общее количество букв в имени и фамилии того, кто описан переменной willie.

9. Определите шаблон структуры, подходящий для хранения следующих элементов: марка автомобиля, его мощность в лошадиных силах, экологический рейтинг, колесная база и год сборки. В качестве дескриптора шаблона используйте car.

10. Предположим, что имеется следующая структура:

```
struct gas {
      float distance;
      float gals;
      float mpg;
};
```

а. Напишите функцию, которая принимает аргумент `struct gas`. Предположите, что передаваемая структура содержит информацию `distance` и `gals`. Эта функция должна вычислять корректное значение для члена `mpg` и возвращать уже заполненную структуру.

б. Напишите функцию, которая принимает в качестве аргумента адрес `struct gas`. Предположите, что передаваемая структура содержит информацию `distance` и `gals`. Эта функция должна вычислять корректное значение для члена `mpg` и присваивать его.

11. Объявите перечисление с дескриптором `choices`, которое устанавливает перечислимые константы `no`, `yes` и `maybe` в `0`, `1` и `2` соответственно.

12. Объявите указатель на функцию, которая возвращает указатель на `char` и принимает в качестве аргументов указатель на `char` и значение `char`.

13. Объявите четыре функции и инициализируйте массив указателей на них. Каждая функция должна принимать два аргумента `double` и возвращать значение `double`. Кроме того, продемонстрируйте два способа использования массива для вызова второй функции с аргументами `10.0` и `2.5`.

Упражнения по программированию

1. Переделайте задание из вопроса 5 таким образом, чтобы аргумент был представлен названием месяца, а не его номером. (Не забывайте о функции `strcmp()`.) Протестируйте готовую функцию в простой программе.

2. Напишите программу, которая предлагает пользователю ввести день, месяц и год. Месяц может быть представлен порядковым номером, названием или аббревиатурой. Затем программа должна возвратить общее количество дней, истекших с начала года по указанный день включительно. (Учитывайте високосные годы.)

3. Измените программу из листинга 14.2, чтобы она сначала выводила описания книг в том порядке, в каком они вводились, затем в алфавитном порядке по названиям и, наконец, в порядке возрастания цены.

4. Напишите программу, которая создает шаблон структуры с двумя членами в соответствии со следующими критериями.

а. Первым членом является номер карточки социального страхования. Второй член — это структура, состоящая из трех членов. Ее первый член содержит имя, второй член — отчество и третий член — фамилию. Создайте и инициализируйте массив из пяти таких структур. Программа должна выводить данные в следующем формате:

```
Dribble, Flossie M. -- 302039823
```

Выводиться должна только начальная буква отчества, за которой следует точка. Разумеется, если этот член пуст, не должен выводиться ни инициал, ни точка. Напишите функцию, которая выполняет такой вывод, передайте рассматриваемую структуру этой функции.

б. Модифицируйте часть а) так, чтобы вместо адреса передавалась сама структура.

5. Напишите программу, которая соответствует следующим требованиям.

а. Программа внешне определяет шаблон структуры name с двумя членами: строкой для хранения имени и строкой для хранения фамилии.

б. Программа внешне определяет шаблон структуры student с тремя членами: структурой name, массивом grade для хранения трех оценок в виде чисел с плавающей запятой и переменной для хранения среднего значения этих трех оценок.

в. Программа содержит функцию main(), где объявляется массив из CSIZE (с CSIZE = 4) структур student, в которых инициализируются члены name именами по вашему выбору. Для выполнения задач, описанных в частях г), д), е) и ж), используйте функции.

г. Программа интерактивно вводит оценки для каждого студента, запрашивая у пользователя ввод имени студента и его оценок. Поместите оценки в массив grade соответствующей структуры. Требуемый цикл можно реализовать в main() или в специальной функции по вашему усмотрению.

д. Программа вычисляет среднюю оценку для каждой структуры и присваивает ее соответствующему члену.

е. Программа выводит информацию из каждой структуры.

ж. Программа выводит среднее значение по курсу для каждого числового члена структуры.

6. Текстовый файл содержит информацию о команде по софтболу (разновидность бейсбола). В каждой строке данные упорядочены следующим образом:

```
4 Джесси Джойбет 5 2 1 1
```

Первым членом является номер игрока (обычно это число из диапазона 0–18). Второй член – это имя игрока, а третий – его фамилия. Каждое имя состоит из одного слова. Следующий член показывает, сколько раз игрок принимал мяч, за которым следует количество нанесенных игроком ударов, проходов и засчитанных пробежек. Файл может содержать результаты более чем одной игры, следовательно, для одного и того же игрока может быть несколько строк. Напишите программу, которая сохраняет соответствующие данные в массиве структур. Структура должна состоять из членов, в которых представлены фамилия и имя, количество набранных очков, проходов и засчитанных пробежек, а также средний результат (эти значения вычисляются позже). В качестве индекса массива можете использовать номер игрока. Программа должна выполнять чтение до конца файла, и накапливать итоговые результаты по каждому игроку.

Мир статистики бейсбола довольно сложен. Например, проход или взятие базы в результате ошибки не расценивается так же высоко, как тот же самый результат, полученный за счет меткого удара, однако он позволяет получить выигранную пробежку. Но эта программа должна только читать и обрабатывать файлы данных, как описано ниже, не заботясь о реалистичности данных.

Простейший способ предусматривает инициализацию содержимого структуры нулями, чтение данных из файла во временные переменные и затем их добавление к содержимому соответствующей структуры. После того, как программа завершит чтение файла, она должна вычислить средний уровень достижений для каждого игрока и запомнить его в соответствующем члене структуры. Средний уровень достижений вычисляется путем деления накопленного числа ударов, выполненных игроком, на количество выходов на ударные позиции; вычисление должно быть с плавающей запятой. Затем программа должна отобразить накапливаемые данные по каждому игроку наряду со строкой, в которой содержатся суммарные статистические данные по всей команде.

7. Модифицируйте код в листинге 14.14 так, чтобы каждая запись читалась из файла и отображалась, чтобы была возможность удалить запись, и можно было изменить ее содержимое. Если вы удаляете запись, используйте освободившуюся позицию массива для чтения следующей записи. Чтобы разрешить изменение существующего содержимого, вместо режима "a+b" необходимо применять "r+b" и уделять больше внимания установке указателя в файле, не допуская перезаписывание существующих записей добавляемыми записями. Проще всего внести все изменения в данные, хранящиеся в памяти, и затем записать всю финальную информацию в файл. Один из возможных подходов к отслеживанию изменений предполагает добавление в структуру каталога члена, который указывает, должен ли он быть удален.

8. Самолетный парк авиакомпании Colossus Airlines включает один самолет с количеством мест 12. Он выполняет один рейс ежедневно. Напишите программу бронирования авиабилетов со следующими характеристиками.

   а. Программа использует массив из 12 структур. Каждая структура содержит идентификационный номер места, специальный маркер, который показывает, забронировано ли место, а также фамилию и имя пассажира, занявшего место.

   б. Программа отображает следующее меню:

   ```
   Для выбора функции введите ее буквенную метку:
   a) Показать количество свободных мест
   b) Показать список свободных мест
   c) Показать список забронированных мест в алфавитном порядке
   d) Забронировать место для пассажира
   e) Снять броню с места
   f) Выйти из программы
   ```

   в. Программа выполняет действия, соответствующие пунктам меню. Позиции меню d и e требуют ввода дополнительных данных, и каждая из них должна позволять прерывать ввод.

   г. По завершении выполнения отдельной функции программа отображает меню снова; исключением является позиция f.

   д. Между запусками программы данные сохраняются в файле. При очередном запуске программа сначала загружает данные из файла, если они есть.

9. Авиакомпания Colossus Airlines (из упражнения 8) приобрела второй самолет (с тем же количеством мест) и расширила обслуживание до четырех рейсов ежедневно (рейсы с номерами 102, 311, 444 и 519). Модифицируйте программу для обработки четырех рейсов. Она должна предлагать меню верхнего уровня, которое позволяет выбирать интересующий рейс и выходить из программы. После выбора рейса должно отобразиться меню, подобное показанному в упражнении 8. Однако в него должен быть добавлен новый элемент — подтверждение брони места. Кроме того, вариант выхода из программы потребуется заменить вариантом возвращения в меню верхнего уровня. При каждом отображении должен указываться номер рейса, обрабатываемого в текущий момент. Вдобавок при отображении брони мест должно выводиться состояние подтверждения.

10. Напишите программу, которая реализует меню с использованием массива указателей на функции. Например, выбор пункта a в меню должен активизировать функцию, на которую указывает первый элемент массива.

11. Напишите функцию по имени transform(), которая принимает четыре аргумента: имя исходного массива, содержащего данные типа double, имя целевого массива типа double, значение int, представляющее количество элементов массива, и имя функции (или, что эквивалентно, указатель на функцию). Функция transform() должна применять указанную функцию к каждому элементу исходного массива и помещать возвращаемое ею значение в целевой массив. Например, вызов

```
transform(source, target, 100, sin);
```

должен установить target[0] в sin(source[0]) и сделать то же самое для 100 элементов массива. Протестируйте функцию в программе, которая вызывает transform() четыре раза, используя в качестве аргументов две функции из библиотеки math.h и две подходящих функции, которые написаны вами специально для transform().

15

Манипулирование битами

В ЭТОЙ ГЛАВЕ...

- Операции: ~, &, |, ^, >>, <<, &=, |=, ^=, >>=, <<=
- Обзор двоичной, восьмеричной и шестнадцатеричной систем счисления
- Два средства языка С для обработки отдельных битов значения: побитовые операции и битовые поля
- Ключевые слова: _Alignas, _Alignof

Язык С позволяет управлять индивидуальными битами значения переменной. Может возникнуть вопрос: для чего это нужно? Не сомневайтесь, что иногда такая возможность необходима или, по крайней мере, удобна. Примером может служить управление некоторым физическим устройством, что часто связано с передачей нескольких битов, причем каждый из них имеет определенный смысл. Кроме того, информация о файлах в операционной системе обычно хранится в виде определенных битов, указывающих на отдельные элементы. Многие операции сжатия и шифрования связаны с управлением битами. Языки высокого уровня, как правило, не обеспечивают такого уровня детализации. Способность совмещать возможности языка высокого уровня с операциями на уровне, который обычно оставляется за языком ассемблера, делает С предпочтительным выбором для написания драйверов устройств и встраиваемого кода.

В этой главе мы исследуем возможности языка С по работе с битами, первоначально ознакомившись с понятиями бита, байта, двоичной и других систем счисления.

Двоичные числа, биты и байты

Обычная форма записи чисел основана на числе 10. Например, число 2157 в позиции тысяч содержит цифру 2, в позиции сотен – 1, в позиции десятков – 5, а в позиции единиц – 7. Это означает, что число 2157 можно рассматривать следующим образом:

```
2 × 1000 + 1 × 100 + 5 × 10 + 7 × 1!
```

Принимая во внимание, что 1000 – это 10 в кубе, 100 – это десять в квадрате, 10 – 10 в первой степени, а 1 – это 10 (как и любое другое положительное число) в нулевой степени, число 2157 можно записать так:

$$2 \times 10^3 + 1 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$$

Поскольку привычная система записи чисел основана на степенях 10, мы говорим, что число 2157 записано *по основанию 10*.

Люди пользуются десятичной системой счисления потому, что у них на руках 10 пальцев. Тогда будем считать, что у бита только два пальца, т.к. он может быть установлен лишь в 0 или 1 (выключен или включен). Таким образом, для компьютера естественной является двоичная система счисления. В ней для записи чисел используются степени 2, а не 10. Числа, выраженные по основанию 2, называют *двоичными*. Число 2 играет такую же роль в двоичной системе, как число 10 в десятичной. Например, двоичная запись 1101 означает:

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

В десятичной записи это становится следующим:

```
1 × 8 + 1 × 4 + 0 × 2 + 1 × 1 = 13
```

В двоичной системе можно представить любое целое число (при достаточном количестве битов) в форме комбинации нулей и единиц. Эта система очень удобна для цифровых вычислительных систем, у которых информация выражается в виде комбинаций включенных и выключенных состояний, что можно интерпретировать как единицы и нули. Давайте посмотрим, как двоичная система работает с однобайтовым целым числом.

Двоичные целые числа

Обычно байт содержит 8 битов. Вспомните, что в языке С термин *байт* применяется для обозначения размера памяти, используемой для хранения набора символов системы, поэтому в С байт может содержать 8, 9, 16 и другое количество битов. Однако в характеристиках модулей памяти и систем передачи данных предполагается, что байт содержит 8 битов. Чтобы излишне не усложнять, в этой главе предполагается 8-битовый байт. (Для ясности в мире вычислений 8-битовый байт часто обозначается термином *октет.*) Можно считать, что биты в байте пронумерованы справа налево с 0 до 7. Седьмой бит называется *старшим*, а нулевой бит — *младшим*. Каждый номер бита соответствует определенной степени числа 2. Такое представление байта иллюстрируется на рис. 15.1.

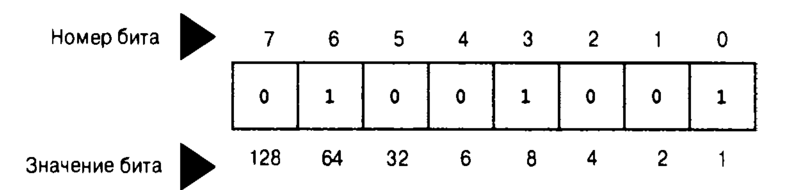


Рис. 15.1. Номера и значения битов

Здесь значение 128 представляет собой 2 в степени 7 и т.д. Байт имеет наибольшее значение, когда все его биты установлены в 1: 11111111. Значение этого двоичного числа определяется следующим образом:

```
128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255
```

Наименьшему значению соответствует комбинация 00000000, или просто 0. Байт может хранить числа от 0 до 255, что составляет 256 возможных значений. Или, интерпретируя комбинацию битов по-другому, программа может применять байт для хранения чисел от -128 до 127, что также дает 256 возможных значений. Например, тип `unsigned char` обычно применяет байт для представления диапазона чисел от 0 до 255, а тип `signed char` — для диапазона от -128 до 127.

Целые числа со знаком

Представление целых чисел со знаком определяется оборудованием, а не языком С. Пожалуй, самый простой способ представления чисел со знаком заключается в резервировании бита, такого как старший, для обозначения знака. В однобайтовом значении для представления самого числа остается 7 битов. В таком представлении *величины со знаком* комбинация 10000001 будет соответствовать числу -1, а комбинация 00000001 — числу 1. Тогда диапазон представляемых значений будет простираться от -127 до +127.

Один из недостатков такого подхода состоит в возможности двоякого представления нуля: +0 и -0. Это вызывает путаницу и приводит к использованию двух комбинаций битов для представления одного значения.

Метод *дополнения до двух* устраняет эту проблему, и в настоящее время он распространен наиболее широко. Мы обсудим его применительно к однобайтовому значению. В данном контексте значения от 0 до 127 представляются последними семью битами со старшим битом, установленным в 0. Пока что нет отличий от представления

величины со знаком. Точно так же, если старший бит равен 1, то число является отрицательным. Отличие начинается при определении значения этого отрицательного числа. Для этого понадобится вычесть комбинацию битов отрицательного числа из 9-битовой комбинации 100000000 (двоичного представления числа 256), в результате получив модуль значения. Для примера предположим, что комбинация имеет вид 10000000. Как байт без знака, это соответствует числу 128. Как значение со знаком, оно является отрицательным (бит 7 равен 1) и имеет величину 100000000–10000000, или 10000000 (т.е. 128). Следовательно, число равно –128. (В представлении величины со знаком оно было бы равно –0.) Подобным же образом, комбинация 10000001 соответствует значению –127, а комбинация 11111111 — значению –1. Данный метод позволяет представлять числа в диапазоне от –128 до 127.

Простейший способ смены знака двоичного числа, которое представлено методом дополнения до 2, предусматривает инвертирование каждого бита (превращение 0 в 1 и 1 в 0) и затем добавление 1. Поскольку 1 — это 00000001, то -1 соответствует 11111110 + 1, или 11111111, как уже было показано.

Метод *дополнения до единицы* формирует отрицательное число путем инвертирования каждого бита в комбинации. Например, комбинация 000000001 — это 1, а 11111110 — значение –1. Этот метод также имеет –0: 11111111. Диапазон представляемых чисел (для однобайтового значения) составляет от –127 до +127.

Двоичные числа с плавающей запятой

Числа с плавающей запятой хранятся в виде двух частей: двоичной дроби и двоичной экспоненты. Давайте посмотрим, как это происходит.

Двоичные дроби

Десятичная дробь 0.527 является следующей суммой:

```
5/10 + 2/100 + 7/1000
```

Здесь знаменатели представляют возрастающие степени 10. В двоичной дроби знаменатели будут степенями 2. Таким образом, двоичная дробь .101 может быть записана так:

```
1/2 + 0/4 + 1/8
```

В десятичной записи это имеет вид:

```
0.50 + 0.00 + 0.125
```

или 0.625.

Многие дроби, такие как 1/3, не могут быть точно представлены в десятичной записи. Аналогично, многие дроби невозможно точно представить и в двоичной записи. На самом деле точно могут быть представлены лишь комбинации составляющих, которые кратны степеням 1/2. Таким образом, дроби 3/4 и 7/8 можно точно записать в двоичном представлении, но дроби 1/3 и 2/5 — нельзя.

Представление чисел с плавающей запятой

Представление числа с плавающей запятой в компьютере предусматривает выделение некоторого количества (в зависимости от системы) битов для хранения двоичной дроби. Дополнительные биты представляют экспоненту. В общих терминах действительное значение числа определяется как произведение двоичной дроби на 2 в степени, выраженной экспонентой. Умножение числа с плавающей запятой, скажем, на 4, увеличивает экспоненту в 2 раза, оставляя двоичную дробь неизменной. Умножение на число, не являющееся степенью 2, изменяет двоичную дробь и при необходимости экспоненту.

Другие основания систем счисления

Специалисты в области компьютеров часто используют системы счисления с основаниями 8 и 16. Поскольку числа 8 и 16 являются степенями 2, эти системы счисления более тесно связаны с двоичной системой компьютера, чем десятичная система.

Восьмеричная система счисления

Восьмеричной называется система счисления с основанием 8. В этой системе каждое знакоместо в числе представляет степень 8. Для записи применяются цифры от 0 до 7. Например, восьмеричное число 451 (в C записывается как 0451) представлено следующим образом:

$$4 \times 8^2 + 5 \times 8^1 + 1 \times 8^0 = 297 \text{ (по основанию 10)}$$

Каждая восьмеричная цифра соответствует трем двоичным цифрам (табл. 15.1). Такое соответствие упрощает перевод чисел между системами. Например, восьмеричное число 0377 – это двоичное число 11111111. Отбросив ведущий 0, мы заменяем 3 комбинацией 011, после чего каждую цифру 7 заменяем 111. Единственное неудобство состоит в том, что трехзначное восьмеричное число в двоичной форме может занимать до 9 битов. Поэтому восьмеричное значение, превышающее 0377, требует более одного байта. Обратите внимание, что внутренние нули не опускаются: числу 0173 соответствует комбинация 01 111 011, а не 01 111 11.

Таблица 15.1. Двоичные эквиваленты восьмеричных цифр

Восьмеричная цифра	Двоичный эквивалент
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Шестнадцатеричная система счисления

Шестнадцатеричной называется система систему счисления с основанием 16. В ней используются степени 16 и цифры от 0 до 15, но из-за того, что в десятичной системе отсутствуют цифры для представления значений от 10 до 15, в шестнадцатеричной системе для них применяются буквы от A до F. Например, шестнадцатеричное число A3F (в C записывается как 0xA3F) представляет следующее значение:

$$10 \times 16^2 + 3 \times 16^1 + 15 \times 16^0 = 2623 \text{ (по основанию 10)}$$

Здесь цифра A представляет значение 10, а F – значение 15. Для обозначения шестнадцатеричных цифр в C разрешено использовать буквы нижнего или верхнего регистра. Таким образом, число 2623 можно записать также в виде 0xa3f.

Каждая шестнадцатеричная цифра соответствует двоичному числу с 4 цифрами, так что две шестнадцатеричных цифры дают в точности один 8-битовый байт. Первая цифра представляет старшие 4 бита, а вторая цифра – младшие 4 бита. Это делает шестнадцатеричное представление естественным выбором для записи значений байтов. Соответствие между шестнадцатеричными цифрами и двоичными числами показано в табл. 15.2. Например, шестнадцатеричное число 0xC2 преобразуется в комбинацию 11000010. Для обратного преобразования комбинацию 11010101 необходимо представить в виде 1101 0101 и затем записать как 0xD5.

Таблица 15.2. Десятичные, шестнадцатеричные числа и их двоичные эквиваленты

Десятичное число	Шестнадцатеричная цифра	Двоичный эквивалент
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Теперь, когда вы ознакомились с понятием битов и байтов, давайте посмотрим, что в языке C можно с ними делать. Существуют два средства, помогающие манипулировать битами. Первое – это набор из шести побитовых операций, которые воздействуют на биты. Второе средство – это форма *полей* данных, которая предоставляет доступ к битам внутри значения `int`. Эти средства обсуждаются в последующих разделах.

Побитовые операции

Язык C предлагает два вида побитовых операций: логические операции и операции сдвига. В последующих примерах мы будем записывать значения в двоичной системе, чтобы вы могли видеть, что происходит с битами. В действительной программе вы будете применять целочисленные переменные или константы в обычных формах. Например, вместо `00011001` будет использоваться запись `25`, `031` или `0x19`. В рассматриваемых примерах мы будем применять 8-битовые числа с нумерацией битов слева направо от 0 до 7.

Побитовые логические операции

Четыре логических побитовых операции работают с целочисленными данными, включая тип `char`. Они называются *побитовыми* потому, что выполняются над каждым битом независимо от бита, находящегося слева или справа. Не путайте их с обычными логическими операциями (`&&`, `||` и `!`), которые имеют дело со значениями целиком.

Дополнение до единицы или побитовое отрицание: ~

Унарная операция ~ преобразует каждую единицу в ноль, а каждый ноль в единицу, как показано в следующем примере:

```
~(10011010)    // выражение
(01100101)     // результат
```

Предположим, что переменной `val` типа `unsigned char` присвоено значение 2. В двоичном виде 2 имеет вид 00000010. Тогда `~val` будет иметь значение 11111101, или 253. Обратите внимание, что операция не изменяет значения переменной `val`, в точности как не изменяет значение `val` выражение `3 * val`; значением `val` по-прежнему является 2, но создается новое значение, которое можно использовать или присваивать где-то в другом месте:

```
newval = ~val;
printf("%d", ~val);
```

Если вы хотите изменить значение `val` на `~val`, применяйте следующий простой оператор присваивания:

```
val = ~val;
```

Побитовая операция "И": &

Двоичная операция `&` создает новое значение за счет выполнения побитового сравнения двух операндов. Для каждой позиции результирующий бит будет равен 1, только если оба соответствующих бита в операндах равны 1. (В терминах истинный/ложный можно сказать, что результат будет истинным, только когда каждый из двух битовых операндов является истинным.) Таким образом, в результате вычисления выражения

```
(10010011) & (00111101)    // выражение
```

получается следующее значение:

```
(00010001)                 // результат
```

Причина в том, что только нулевой и четвертый биты равны 1 в обоих операндах. В С также имеется операция "И", объединенная с присваиванием: `&=`.

Оператор

```
val &= 0377;
```

дает такой же результат, как и следующий оператор:

```
val = val & 0377;
```

Побитовая операция "ИЛИ": |

Двоичная операция `|` создает новое значение за счет выполнения побитового сравнения двух операндов. Для каждой позиции бит будет равен 1, если любой из соответствующих битов в операндах равен 1. (В терминах истинный/ложный можно сказать, что результат будет истинным в случае, когда один или другой битовый операнд является истинным либо сразу оба.)

Таким образом, в результате вычисления выражения

```
(10010011) | (00111101)    // выражение
```

получается следующее значение:

```
(10111111)                 // результат
```

Причина в том, что биты во всех позициях кроме 6 имеют значение 1 в одном или в другом операнде (или в обоих). В С также существует операция "ИЛИ", объединенная с присваиванием: |=. Оператор

```
val |= 0377;
```

дает тот же результат, что и следующий оператор:

```
val = val | 0377;
```

Побитовое "исключающее ИЛИ": ^

Двоичная операция ^ выполняет побитовое сравнение двух операндов. Для каждой позиции результирующий бит будет равен 1, если один или другой (но не оба) из соответствующих битов в операндах равен 1. (В терминах истинный/ложный можно сказать, что результат будет истинным в случае, когда один или другой битовый операнд является истинным, но не оба.) Таким образом, в результате вычисления выражения

```
(10010011) ^ (00111101)    // выражение
```

получается следующее значение:

```
(10101110)                 // результат
```

Обратите внимание, что поскольку бит 0 равен 1 у обоих операндов, результирующий бит 0 получает значение 0.

В языке С также имеется операция "исключающее ИЛИ", объединенная с присваиванием: ^=. Оператор

```
val ^= 0377;
```

дает тот же результат, что и следующий оператор:

```
val = val ^ 0377;
```

Случай применения: маски

Побитовая операция "И" часто используется с маской. *Маска* – это комбинация битов, в которой некоторые биты включены (1), а некоторые выключены (0). Чтобы понять, почему ее так назвали, давайте посмотрим, что происходит, когда мы объединяем какую-то величину с маской с применением операции &. Для примера предположим, что вы определили символическую константу MASK как 2 (т.е. 00000010), у которой ненулевым является только бит с номером 1. Тогда оператор

```
flags = flags & MASK;
```

приведет к установке всех битов flags (кроме первого) в 0, т.к. любой бит, объединяемый с 0 посредством операции "И", дает 0. Бит номер 1 переменной остается неизменным. (Если бит равен 1, то значением 1 & 1 будет 1; если же бит равен 0, то 0 & 1 дает 0.) Такой процесс называется "использованием маски", поскольку нули в маске скрывают соответствующие биты в переменной flags.

Развивая аналогию, биты с 0 в маске можно считать непрозрачными, а биты с 1 – прозрачными. Выражение flags & MASK похоже на накрывание маской комбинации битов flags; видимыми из-под маски будут только те биты, которым в MASK соответствуют биты с 1 (рис. 15.2).

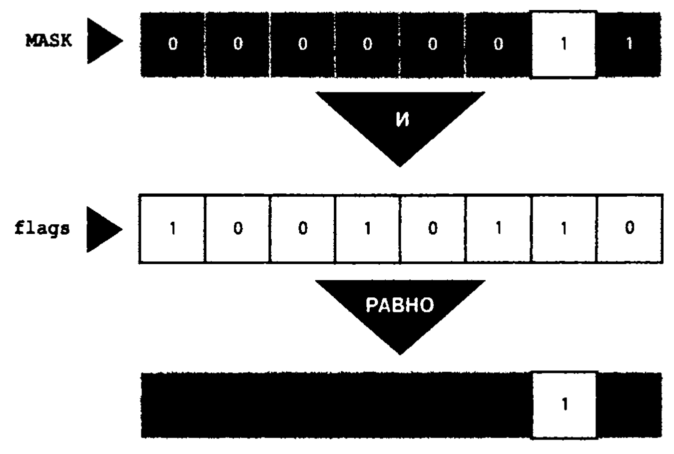


Рис. 15.2. Наглядное представление маски

Для сокращения кода можно применить операцию "И", объединенную с присваиванием:

```
flags &= MASK;
```

Ниже показан один распространенный случай использования этой операции:

```
ch &= 0xff;  /* или ch &= 0377; */
```

Вспомните, что значение `0xff` записывается как `11111111` в двоичном или как `0377` в восьмеричном виде. Эта маска оставляет последние восемь битов в `ch` без изменений, а остальные устанавливает в 0. Независимо от того, сколько битов содержит исходная переменная `ch` – 8, 16 или более, – финальное значение усекается до величины, которая умещается в один 8-битовый байт. В данном случае маска имеет ширину 8 битов.

Случай применения: включение (установка) битов

Иногда требуется включить отдельные биты в значении, оставив остальные без изменений. Например, компьютер IBM PC управляет оборудованием, отправляя нужные значения в порты. Для активизации, скажем, динамика, необходимо включить бит 1, а остальные биты оставить неизменными. Этого можно сделать с помощью побитовой операции "ИЛИ".

Например, пусть имеется константа `MASK`, в которой бит 1 установлен в 1. Тогда оператор

```
flags = flags | MASK;
```

включает бит номер 1 в переменной `flags` и оставляет все остальные биты без изменений. Это объясняется тем, что любой бит, объединенный с 0 посредством операции `|`, остается самим собой, а объединенный с 1 с использованием `|`, становится равным 1.

Например, пусть `flags` равно `00001111` и `MASK` – `10110110`. Выражение

```
flags | MASK
```

становится

```
(00001111) | (10110110)     // выражение
```

и после вычисления дает следующий результат:

```
(10111111)                  // результат
```

Все биты, установленные в 1 внутри MASK, также будут установлены в 1 в результате. Все биты в flags, которые соответствуют битам 0 в MASK, остаются неизмененными.

Для краткости можно использовать побитовую операцию "ИЛИ", объединенную с присваиванием:

```
flags |= MASK;
```

Этот оператор установит в 1 те биты falgs, которым соответствуют включенные биты в MASK, оставив другие биты без изменений.

Случай применения: выключение (очистка) битов

Точно так же, как удобно иметь возможность включать отдельные биты, не затрагивая остальные, не менее удобно располагать возможностью их выключения. Предположим, что требуется отключить бит номер 1 в переменной flags. И снова MASK имеет включенный только бит 1. Можно воспользоваться следующим оператором:

```
flags = flags & ~MASK;
```

Поскольку в MASK все биты кроме бита 1 выключены, выражение ~MASK дает значение, в котором все биты кроме бита 1 включены. Объединение 1 с любым битом, используя операцию &, дает сам этот бит, поэтому оператор оставляет все биты кроме бита 1 без изменений. Объединение 0 с любым битом посредством операции & дает 0 независимо от исходного значения бита.

Например, пусть flags равно 00001111 и MASK — 10110110. Выражение

```
flags & ~MASK
```

становится

```
(00001111) &^ (10110110)        // выражение
```

и после вычисления дает следующий результат:

```
(00001001)                      // результат
```

Все биты, установленные в 1 внутри MASK, будут установлены в 0 в результате. Все биты в flags, которые соответствуют битам 0 в MASK, остаются неизмененными.

Ниже представлена сокращенная форма:

```
flags &= ~MASK;
```

Случай применения: переключение битов

Переключение бита означает его выключение, если он включен, и включение, если выключен. Для переключения битов можно применять побитовую операцию исключающего "ИЛИ". Идея в том, что если b — это установленное состояние бита (1 или 0), то 1 ^ b равно 0, когда b равно 1, и 1, когда b равно 0. Кроме того, выражение 0 ^ b дает b независимо от значения b. Следовательно, в результате объединения значения с маской с использованием операции ^ биты, соответствующие 1 в маске, переключаются, а биты, соответствующие 0 в маске, останутся неизмененными. Чтобы переключить бит 1 переменной flags, можно выполнить одно из следующих действий:

```
flags = flags ^ MASK;
flags ^= MASK;
```

Например, пусть flags равно 00001111 и MASK — 10110110. Выражение

```
flags ^ MASK
```

становится

```
(00001111) ^ (10110110)  // выражение
```

и после вычисления дает следующий результат:

```
(10111001)                // результат
```

Все биты, установленные в 1 внутри MASK, приводят к переключению соответствующих битов в flags. Все биты в flags, которые соответствуют битам 0 в MASK, остаются неизмененными.

Случай применения: проверка значения бита

Вы уже видели, как изменять значения битов. Предположим, что вместо этого нужно проверить значение какого-нибудь бита. Например, установлен ли в 1 бит 1 в flags? Простое сравнение flags и MASK здесь не подойдет:

```
if (flags == MASK)
   puts("Совпадает!");    /* не работает */
```

Даже если бит 1 переменной flags установлен в 1, значение какого-то другого бита в flags может сделать результат сравнения недействительным. Чтобы выполнить сравнение только бита 1 в flags с MASK, необходимо сначала замаскировать остальные биты flags:

```
if ((flags & MASK) == MASK)
   puts("Совпадает!");
```

Побитовые операции имеют приоритет ниже, чем у операции ==, поэтому выражение flags & MASK должно быть заключено в скобки.

Во избежание неполного охвата информации, битовая маска должна иметь ширину не меньше, чем у маскируемого значения.

Побитовые операции сдвига

Теперь давайте взглянем на операции сдвига в языке С. Побитовые операции сдвига сдвигают биты влево или вправо. Для большей наглядности мы здесь также будем применять двоичную запись чисел.

Сдвиг влево: <<

Операция сдвига влево (<<) сдвигает биты значения левого операнда влево на количество позиций, заданное правым операндом. Освобождаемые позиции заполняются 0, а биты, выходящие за пределы значения левого операнда, теряются. В следующем примере каждый бит сдвигается на две позиции влево:

```
(10001010) << 2          // выражение
(00101000)               // результат
```

Эта операция выдает новое битовое значение, но не изменяет операнды. Для примера предположим, что переменная stonk имеет значение 1. Выражение stonk<<2 дает 4, но значением stonk по-прежнему является 1. Чтобы изменить значение переменной, можно воспользоваться операцией сдвига влево с присваиванием (<<=). Эта операция сдвигает биты переменной влево на количество позиций, указанное в правом операнде. Вот пример:

```
int stonk = 1;
int onkoo;
onkoo = stonk << 2;       /* присваивает 4 переменной onkoo */
stonk <<= 2;              /* изменяет значение stonk на 4   */
```

Сдвиг вправо: >>

Операция сдвига вправо (>>) сдвигает биты значения левого операнда вправо на количество позиций, указанное в правом операнде. Биты, которые выходят за правую границу левого операнда, теряются. Для типов без знака освобождаемые слева позиции заполняются 0. Для типов со знаком данных результат зависит от системы. Освобождаемые позиции могут заполняться 0 либо битом знака (самого левого):

```
(10001010) >> 2      // выражение, значение со знаком
(00100010)           // результат в одних системах
(10001010) >> 2      // выражение, значение со знаком
(11100010)           // результат в других системах
```

Для значения без знака результат будет следующим:

```
(10001010) >> 2      // выражение, значение без знака
(00100010)           // результат во всех системах
```

Каждый бит перемещается на две позиции вправо, а освобождаемые позиции заполняются 0.

Операция сдвига вправо с присваиванием (>>=) сдвигает вправо биты левого операнда на заданное в правом операнде количество позиций, например:

```
int sweet = 16;
int ooosw;
ooosw = sweet >> 3;  /* ooosw равно 2, sweet по-прежнему 16 */
sweet >>=3;          /* значение sweet изменилось на 2      */
```

Случай применения: побитовые операции сдвига

Побитовые операции сдвига могут служить удобным и эффективным (в зависимости от оборудования) средством выполнения умножения и деления на степени 2:

`number << n` Умножает `number` на 2 в степени `n`

`number >> n` Делит `number` на 2 в степени `n`, если значение `number` неотрицательно

Эти операции сдвига аналогичны смещению десятичной точки при умножении или делении на 10.

Операции сдвига могут также использоваться для извлечения групп битов из более крупных конструкций. Предположим, что для представления значений цвета применяется переменная типа `unsigned long`, причем младший байт содержит интенсивность красной составляющей, следующий байт — интенсивность зеленой составляющей, а третий байт — интенсивность синей составляющей цвета. Пусть необходимо сохранить интенсивность каждой составляющей в собственной переменной типа `unsigned char`. Для этого можно написать такой код:

```
#define BYTE_MASK 0xff
unsigned long color = 0x002a162f;
unsigned char blue, green, red;
red = color & BYTE_MASK;
green = (color >> 8) & BYTE_MASK;
blue = (color >> 16) & BYTE_MASK;
```

В коде посредством операции сдвига вправо 8-битовое значение составляющей цвета перемещается в младший байт. Затем с помощью приема с маской значение младшего байта присваивается желаемой переменной.

Пример программы

В главе 9 при написании программы преобразования чисел в двоичное представление мы использовали рекурсию. Теперь мы решим ту же задачу с применением побитовых операций. Программа в листинге 15.1 читает вводимое с клавиатуры целое число и передает его вместе с адресом строки в функцию по имени `itobs()`, которая строит для целочисленного значения строку с двоичным представлением. Для определения подходящей комбинации 0 и 1, помещаемой в строку, эта функция использует побитовые операции.

Листинг 15.1. Программа `binbit.c`

```
/* binbit.c -- использование операций с битами для отображения двоичного
представления чисел */
#include <stdio.h>
#include <limits.h>  // для CHAR_BIT количество битов на символ
char * itobs(int, char *);
void show_bstr(const char *);

int main(void)
{
   char bin_str[CHAR_BIT * sizeof(int) + 1];
   int number;

   puts("Вводите целые числа и просматривайте их двоичные представления.");
   puts("Нечисловой ввод завершает программу.");
   while (scanf("%d", &number) == 1)
   {
      itobs(number,bin_str);
      printf("%d представляется как ", number);
      show_bstr(bin_str);
      putchar('\n');
   }
   puts("Программа завершена.");

   return 0;
}

char * itobs(int n, char * ps)
{
   int i;
   const static int size = CHAR_BIT * sizeof(int);

   for (i = size - 1; i >= 0; i--, n >>= 1)
      ps[i] = (01 & n) + '0';  // предполагается кодировка ASCII или похожая
   ps[size] = '\0';

   return ps;
}

/* отображение двоичной строки блоками по 4 */
void show_bstr(const char * str)
{
   int i = 0;

   while (str[i])  /* пока не будет получен нулевой символ */
   {
      putchar(str[i]);
      if(++i % 4 == 0 && str[i])
         putchar(' ');
   }
}
```

В листинге 15.1 применяется макрос CHAR_BIT из заголовочного файла limits.h. Этот макрос представляет количество битов в типе char. Операция sizeof возвращает размер в терминах char, поэтому выражение CHAR_BIT * sizeof(int) дает количество битов в значении int. Массив bin_str содержит на один элемент больше этой величины, чтобы можно было добавить в него завершающий нулевой символ.

Функция itobs() возвращает тот же самый адрес, который ей был передан, так что ее вызов можно использовать, к примеру, в качестве аргумента printf(). На первой итерации цикла for функция вычисляет выражение 01 & n. Операнд 01 – это восьмеричное представление маски, у которой все биты кроме нулевого установлены в 0. Следовательно, результатом 01 & n будет значение последнего бита в n. Значением является 0 или 1, но для массива необходим *символ* '0' или *символ* '1'. Преобразование осуществляется добавлением кода для '0'. (Это предполагает, что цифры кодируются последовательно, как в ASCII.) Результат помещается в предпоследний элемент массива. (Последний элемент зарезервирован для нулевого символа.)

Кстати, вместо выражения 01 & n можно применить и 1 & n. Использование восьмеричного значения 1 вместо десятичного выглядит более стильно. С этой точки зрения вариант 0x1 & n, пожалуй, даже лучше.

Затем в цикле выполняются операторы i-- и n >>= 1. Первый оператор приводит к переходу на предыдущий элемент массива, а второй сдвигает биты в n на одну позицию вправо. На следующей итерации цикла код найдет значение нового самого правого бита. После этого соответствующий ему символ цифры помещается в элемент, предшествующий последней цифре. В подобной манере функция заполняет массив справа налево.

Для отображения результирующей строки можно применять printf() или puts(). Тем не менее, в листинге 15.1 определена функция show_bstr(), которая разбивает последовательность битов на группы по четыре, чтобы облегчить восприятие строки.

Ниже приведен пример выполнения программы:

```
Вводите целые числа и просматривайте их двоичные представления.
Нечисловой ввод завершает программу.
7
7 представляется как 0000 0000 0000 0000 0000 0000 0000 0111
2013
2013 представляется как 0000 0000 0000 0000 0000 0111 1101 1101
-1
-1 is 1111 1111 1111 1111 1111 1111 1111 1111
32123
32123 представляется как 0000 0000 0000 0000 0111 1101 0111 1011
q
Программа завершена.
```

Еще один пример

Давайте рассмотрим еще один пример. На этот раз цель заключается в том, чтобы написать функцию, которая инвертирует последние n битов в значении, принимая в качестве аргументов n и само значение.

Операция ~ инвертирует биты, но делает это со всеми битами в байте, а не только с избранными. Однако, как вы уже видели, для переключения отдельных битов можно использовать операцию ^ (исключающее "ИЛИ"). Предположим, что создана маска, в которой последние n битов установлены в 1, а остальные – в 0. Тогда применение ^ к этой маске и значению переключает, или *инвертирует*, последние n битов, оставляя остальные биты без изменений. Такой подход реализован в следующем фрагменте кода:

```
int invert_end(int num, int bits)
{
    int mask = 0;
    int bitval = 1;
    while (bits-- > 0)
    {
        mask |= bitval;
        bitval <<= 1;
    }
    return num ^ mask;
}
```

Маска создается в цикле `while`. Изначально в `mask` все биты установлены в 0. На первой итерации цикла бит 0 устанавливается в 1, после чего значение `bitval` увеличивается до 2, т.е. в нем бит 0 устанавливается в 0, а бит 1 — в 1. На следующей итерации бит 1 в `mask` устанавливается в 1 и т.д. В конце концов, операция `num ^ mask` дает желаемый результат.

Для тестирования функции ее можно внедрить в предыдущую программу, как показано в листинге 15.2.

Листинг 15.2. Программа `invert4.c`

```
/* invert4.c -- использование операций с битами для отображения двоичного
представления чисел */
#include <stdio.h>
#include <limits.h>
char * itobs(int, char *);
void show_bstr(const char *);
int invert_end(int num, int bits);
int main(void)
{
   char bin_str[CHAR_BIT * sizeof(int) + 1];
    int number;
    puts("Вводите целые числа и просматривайте их двоичные представления.");
    puts("Нечисловой ввод завершает программу.");
    while (scanf("%d", &number) == 1)
    {
        itobs(number,bin_str);
        printf("%d представляется как\n", number);
        show_bstr(bin_str);
        putchar('\n');
        number = invert_end(number, 4);
        printf("Инвертирование последних 4 битов дает\n");
        show_bstr(itobs(number,bin_str));
        putchar('\n');
    }
    puts("Программа завершена.");
    return 0;
}
char * itobs(int n, char * ps)
{
    int i;
    const static int size = CHAR_BIT * sizeof(int);
    for (i = size - 1; i >= 0; i--, n >>= 1)
        ps[i] = (01 & n) + '0';
    ps[size] = '\0';
    return ps;
}
```

```
/* отображение двоичной строки блоками по 4 */
void show_bstr(const char * str)
{
   int i = 0;
   while (str[i])  /* пока не будет получен нулевой символ */
   {
      putchar(str[i]);
      if(++i % 4 == 0 && str[i])
         putchar(' ');
   }
}
int invert_end(int num, int bits)
{
   int mask = 0;
   int bitval = 1;
   while (bits-- > 0)
   {
      mask |= bitval;
      bitval <<= 1;
   }
   return num ^ mask;
}
```

Ниже представлен пример выполнения программы:

```
Вводите целые числа и просматривайте их двоичные представления.
Нечисловой ввод завершает программу.
7
7 представляется как
0000 0000 0000 0000 0000 0000 0000 0111
Инвертирование последних 4 битов дает
0000 0000 0000 0000 0000 0000 0000 1000
12541
12541 представляется как
0000 0000 0000 0000 0011 0000 1111 1101
Инвертирование последних 4 битов дает
0000 0000 0000 0000 0011 0000 1111 0010
q
Bye!
```

Битовые поля

Второй метод манипулирования битами предусматривает использование *битового поля*, которое представляет собой просто набор соседствующих битов внутри значения типа signed int или unsigned int. (Стандарты C99 и C11 дополнительно разрешают иметь битовые поля типа _Bool.) Битовое поле создается путем объявления структуры, в которой помечено каждое поле и определен его размер. Например, следующее объявление устанавливает четыре однобитовых поля:

```
struct {
   unsigned int autfd   : 1;
   unsigned int bldfc   : 1;
   unsigned int undln   : 1;
   unsigned int itals   : 1;
} prnt;
```

Такое определение приводит к получению структуры `prnt`, содержащей четыре однобитовых поля. Теперь для присваивания значений отдельным полям можно применять обычную операцию членства в структуре:

```
prnt.itals = 0;
prnt.undln = 1;
```

Поскольку каждое из этих полей – это просто один бит, присваивать можно только значения 1 и 0. Переменная `prnt` хранится в ячейке памяти размером типа `int`, но в этом примере используются только четыре бита.

Структуры с битовыми полями служат удобным средством для отслеживания настроек. Многие настройки, такие как полужирное или курсивное начертание шрифта, сводятся к указанию одной из двух опций: "включено" или "отключено", "да" или "нет", "истинно" или "ложно". Когда нужен одиночный бит, не имеет смысла применять целую переменную. Структура с битовыми полями позволяет хранить множество настроек в одной конструкции.

Временами настройка предусматривает более двух опций, поэтому для представления всех вариантов одного бита оказывается недостаточно. Это не проблема, т.к. размеры полей не ограничены одним битом. Структуру можно определить следующим образом:

```
struct {
   unsigned int code1 : 2;
   unsigned int code2 : 2;
   unsigned int code3 : 8;
} prcode;
```

Этот код создает два 2-битовых поля и одно 8-битовое. Теперь возможны следующие присваивания:

```
prcode.code1 = 0;
prcode.code2 = 3;
prcode.code3 = 102;
```

Нужно просто следить, чтобы значение не превышало размерность поля.

А что, если общее количество объявленных битов превысит размер типа `unsigned int`? Тогда будет использоваться следующая область для хранения `unsigned int`. Отдельное поле не должно перекрывать границу между двумя смежными областями `unsigned int`. Компилятор автоматически сдвигает такое перекрывающее определение поля, чтобы выровнять его по границе `unsigned int`. Когда это происходит, в первой области `unsigned int` остается неименованный промежуток.

Структуру полей можно заполнить неименованными промежутками с применением ширин неименованных полей. Использование неименованного поля шириной 0 приводит к тому, что следующее поле выравнивается по следующей области целочисленного значения:

```
struct {
   unsigned int field1 : 1;
   unsigned int        : 2;
   unsigned int field2 : 1;
   unsigned int        : 0;
   unsigned int field3 : 1;
} stuff;
```

Здесь между полями `stuff.field1` и `stuff.field2` имеется 2-битовый промежуток, а поле `stuff.field3` хранится в следующей области `int`.

Важной зависимостью от системы является порядок, в котором поля помещаются в область int. В одних системах поддерживается порядок слева направо, в других – справа налево. Кроме того, системы различаются местоположением границ между полями. По этим причинам битовые поля не особенно переносимы. Однако обычно они применяются в целях, не предполагающих переносимость, таких как размещение данных в точной форме, используемой отдельным аппаратным устройством.

Пример с битовыми полями

Битовые поля часто применяются в качестве более компактного способа хранения данных. Предположим, что вы решили представить свойства выводимого на экран окна. Давайте отложим в сторону все сложности графики и предположим, что окно обладает только перечисленными ниже свойствами.

- Окно может быть прозрачным или непрозрачным.
- Цвет фона выбирается из следующей палитры: черный, красный, зеленый, желтый, синий, пурпурный, голубой и белый.
- Рамка может быть скрыта или отображена.
- Цвет рамки выбирается из той же палитры, что и цвет фона.
- Для рамки применяются три стиля линии: сплошная, пунктирная и штриховая.

Для каждого свойства можно было бы использовать отдельную переменную или полноразмерный член структуры, но это привело бы к напрасному расходу битов. Например, для указания прозрачности или непрозрачности окна достаточно одного бита. То же самое можно сказать о свойстве отображения или сокрытия рамки. Восемь возможных значений цвета могут быть представлены 3-битовым элементом, а 2-битового элемента более чем достаточно для представления трех возможных стилей рамки. Таким образом, для представления всех пяти свойств достаточно 10 битов.

Один из вариантов представления информации предусматривает применение заполнителей, чтобы поместить связанную с фоном окна информацию в один байт, а связанную с рамкой – во второй. Это реализовано в следующем объявлении struct box_props:

```
struct box_props {
    bool opaque                 : 1;
    unsigned int fill_color     : 3;
    unsigned int                : 4;
    bool show_border            : 1;
    unsigned int border_color   : 3;
    unsigned int border_style   : 2;
    unsigned int                : 2;
};
```

В результате использования заполнителей размер структуры увеличивается до 16 битов. Без них было бы достаточно 10 битов. Однако имейте в виду, что в C для структур с битовыми полями в качестве базовой единицы размещения применяется тип unsigned int. Поэтому, даже если структура содержит единственный элемент, которым является однобитовое поле, структура будет иметь такой же размер, как у типа unsigned int, что в нашей системе составляет 32 бита. Кроме того, в этом коде предполагается, что тип _Bool из C99 доступен и в заголовочном файле stdbool.h ему назначен псевдоним bool.

Для члена opaque можно использовать значение 1 для указания непрозрачности окна и значение 0 – для прозрачности. То же самое применимо к члену show_border.

Для цветов можно использовать простое представление RGB (красный, зеленый, синий). Это основные цвета для смешивания спектра. В мониторе для воспроизведения различных цветов применяется смешанное свечение красных, зеленых и синих пикселей. В ранних моделях мониторов каждый пиксель мог иметь только включенное или выключенное состояние, поэтому для представления интенсивности каждой из трех составляющих было достаточно одного бита. Обычно левый бит представлял интенсивность синего, средний – интенсивность зеленого, а правый – красного цвета. В табл. 15.3 показаны восемь возможных комбинаций. Они могут служить значениями для членов `fill_color` и `border_color`. Наконец, значения 0, 1 и 2 могут представлять сплошной, пунктирный и штриховой тип линий, определяемый членом `border_style`.

Таблица 15.3. Простое представление цветов

Комбинация битов	Десятичный эквивалент	Цвет
000	0	Черный
001	1	Красный
010	2	Зеленый
011	3	Желтый
100	4	Синий
101	5	Пурпурный
110	6	Голубой
111	7	Белый

В листинге 15.3 структура `box_props` используется в простом примере. Директивы `#define` применяются для создания символических констант, представляющих возможные значения членов. Обратите внимание, что основные цвета представлены включением единственного бита. Остальные цвета могут представляться комбинациями основных цветов. Например, пурпурный цвет создается включением битов синего и красного цветов, поэтому его можно записать как комбинацию `BLUE | RED`.

Листинг 15.3. Программа `fields.c`

```
/* fields.c — определение и использование полей */
#include <stdio.h>
#include <stdbool.h>  // C99, определение bool, true, false
/* стили линии    */
#define SOLID  0
#define DOTTED 1
#define DASHED 2
/* основные цвета  */
#define BLUE   4
#define GREEN  2
#define RED    1
/* смешанные цвета */
#define BLACK  0
#define YELLOW  (RED | GREEN)
#define MAGENTA (RED | BLUE)
#define CYAN    (GREEN | BLUE)
#define WHITE   (RED | GREEN | BLUE)

const char * colors[8] = {"черный", "красный", "зеленый", "желтый",
    "синий", "пурпурный", "голубой", "белый"};
```

```
struct box_props {
   bool opaque                 : 1;      // или unsigned int (до C99)
   unsigned int fill_color     : 3;
   unsigned int                : 4;
   bool show_border            : 1;      // или unsigned int (до C99)
   unsigned int border_color   : 3;
   unsigned int border_style   : 2;
   unsigned int                : 2;
};
void show_settings(const struct box_props * pb);
int main(void)
{
   /* создание и инициализация структуры box_props */
   struct box_props box = {true, YELLOW , true, GREEN, DASHED};

   printf("Исходные настройки окна:\n");
   show_settings(&box);

   box.opaque = false;
   box.fill_color = WHITE;
   box.border_color = MAGENTA;
   box.border_style = SOLID;
   printf("\nИзмененные настройки окна:\n");
   show_settings(&box);

   return 0;
}
void show_settings(const struct box_props * pb)
{
   printf("Окно %s.\n",
         pb->opaque == true ? "непрозрачно": "прозрачно");
   printf("Цвет фона %s.\n", colors[pb->fill_color]);
   printf("Рамка %s.\n",
         pb->show_border == true ? "отображается" : "не отображается");
   printf("Цвет рамки %s.\n", colors[pb->border_color]);
   printf ("Стиль рамки ");
   switch(pb->border_style)
   {
      case SOLID  : printf("сплошной.\n"); break;
      case DOTTED : printf("пунктирный.\n"); break;
      case DASHED : printf("штриховой.\n"); break;
      default     : printf("неизвестного типа.\n");
   }
}
```

Вот вывод, полученный из программы:

```
Исходные настройки окна:
Окно непрозрачно.
Цвет фона желтый.
Рамка отображается.
Цвет рамки зеленый.
Стиль рамки штриховой.

Измененные настройки окна:
Окно прозрачно.
Цвет фона белый.
Рамка отображается.
Цвет рамки пурпурный.
Стиль рамки сплошной.
```

Отметим несколько моментов. Прежде всего, структуру битовых полей можно инициализировать с использованием обычного для структур синтаксиса:

```
struct box_props box = {YES, YELLOW , YES, GREEN, DASHED};
```

Аналогично можно присваивать значения элементам битовых полей:

```
box.fill_color = WHITE;
```

Кроме того, член битового поля может служить выражением в операторе switch. Он даже может выступать в качестве индекса массива:

```
printf("Цвет фона %s.\n", colors[pb->fill_color]);
```

Обратите внимание, что массив colors был определен так, чтобы каждое значение индекса соответствовало строковому представлению названия цвета, имеющего значение индекса, которое совпадает с числовым значением цвета. Например, индекс 1 соответствует строке "красный" и константа RED имеет значение 1.

Битовые поля и побитовые операции

Битовые поля и побитовые операции — это два альтернативных подхода к решению задачи программирования одного и того же типа. Это значит, что часто можно применять любой из подходов. В предыдущем примере для хранения информации о графическом окне использовалась структура с размером, как у типа unsigned int. Ту же самую информацию можно было бы сохранить в переменной типа unsigned int. Затем вместо синтаксиса членства в структуре можно было бы применять побитовые операции. Обычно такая методика не очень удобна. Рассмотрим пример, в котором задействованы оба подхода. (Оба подхода здесь применяются для иллюстрации отличий между ними, а вовсе не из-за того, чтобы внушить мысль о целесообразности их одновременного использования!)

В качестве средства комбинирования подхода на основе структуры и подхода на базе побитовых операций можно воспользоваться объединением. Исходя из существующего объявления типа struct box_props, можно объявить следующее объединение:

```
union Views    /* взгляд на данные как на struct или как на unsigned short */
{
    struct box_props st_view;
    unsigned short us_view;
};
```

В некоторых системах переменная unsigned short и структура box_props занимают 16 битов в памяти. В других системах, таких как наша, unsigned short и box_props занимают 32 бита. В любом случае это объединение позволяет применять член st_view, чтобы трактовать отведенную память как структуру, или использовать член us_view, чтобы рассматривать тот же самый блок памяти как значение unsigned short. Какие битовые поля структуры соответствуют отдельным битам переменной типа unsigned short? Это зависит от реализации и оборудования. В следующем примере предполагается, что структуры загружены в память, начиная с младших битов и заканчивая старшими битами байта. Другими словами, первое битовое поле в структуре соответствует биту 0 слова. (Для простоты эта идея иллюстрируется на рис. 15.3 для 16-битового единицы.)

В листинге 15.4 объединение Views применяется для сравнения подходов на основе битовых полей и побитовых операций. Здесь box — это объединение Views, поэтому box.st_view представляет собой структуру box_props, использующую битовые поля, а box.ui_view — те же самые данные, но представленные как значение unsigned short.

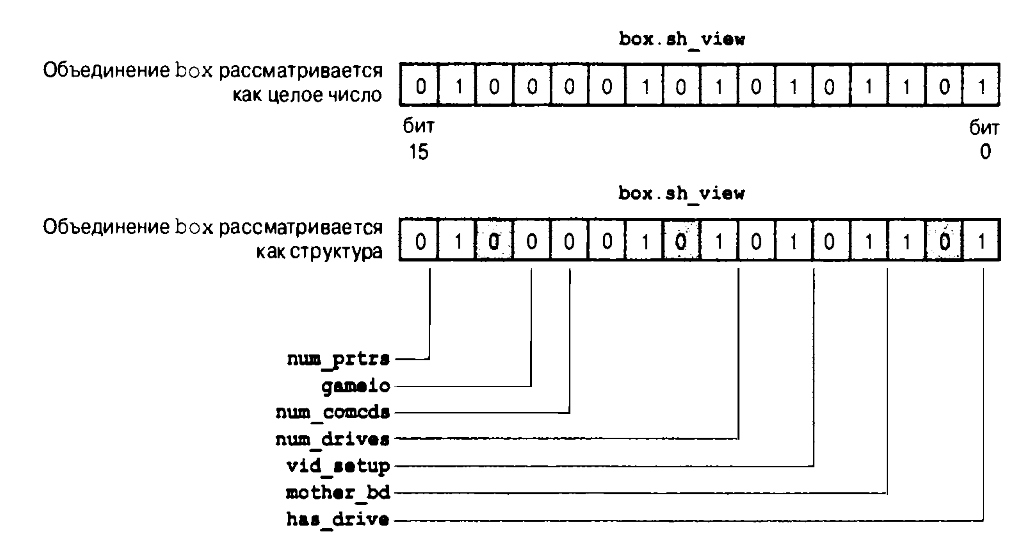


Рис. 15.3. Объединение как целое число и как структура

Вспомните, что объединение может иметь инициализированный первый член, поэтому установленные значения соответствуют представлению структуры. Программа отображает свойства окна с помощью функции, основанной на представлении структуры, и также посредством функции, основанной на представлении unsigned short. Любой из подходов обеспечивает доступ к данным, но по-разному. Вдобавок в программе применяется определенная ранее в главе функция itobs(), которая позволяет отобразить данные в виде строки двоичных цифр, чтобы можно было видеть, какие биты включены, а какие выключены.

Листинг 15.4. Программа dualview.c

```
/* dualview.c -- битовые поля и побитовые операции */
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
/* КОНСТАНТЫ БИТОВЫХ ПОЛЕЙ */
/* стили линии    */
#define SOLID  0
#define DOTTED 1
#define DASHED 2
/* основные цвета  */
#define BLUE   4
#define GREEN  2
#define RED    1
/* смешанные цвета */
#define BLACK  0
#define YELLOW  (RED | GREEN)
#define MAGENTA (RED | BLUE)
#define CYAN    (GREEN | BLUE)
#define WHITE   (RED | GREEN | BLUE)

/* ПОБИТОВЫЕ КОНСТАНТЫ   */
#define OPAQUE          0x1
#define FILL_BLUE       0x8
#define FILL_GREEN      0x4
```

```
#define FILL_RED         0x2
#define FILL_MASK        0xE
#define BORDER           0x100
#define BORDER_BLUE      0x800
#define BORDER_GREEN     0x400
#define BORDER_RED       0x200
#define BORDER_MASK      0xE00
#define B_SOLID          0
#define B_DOTTED         0x1000
#define B_DASHED         0x2000
#define STYLE_MASK       0x3000

const char * colors[8] = {"черный", "красный", "зеленый", "желтый",
   "синий", "пурпурный", "голубой", "белый"};
struct box_props {
   bool opaque                : 1;
   unsigned int fill_color    : 3;
   unsigned int               : 4;
   bool show_border           : 1;
   unsigned int border_color  : 3;
   unsigned int border_style  : 2;
   unsigned int               : 2;
};

union Views    /* взгляд на данные как на struct или как на unsigned short */
{
   struct box_props st_view;
   unsigned short us_view;
};

void show_settings(const struct box_props * pb);
void show_settings1(unsigned short);
char * itobs(int n, char * ps);

int main(void)
{
   /* создание объекта Views, инициализация представления в виде структуры */
   union Views box = {{true, YELLOW , true, GREEN, DASHED}};
   char bin_str[8 * sizeof(unsigned int) + 1];

   printf("Исходные настройки окна:\n");
   show_settings(&box.st_view);
   printf("\nНастройки окна с использованием представления unsigned short:\n");
   show_settings1(box.us_view);

   printf("комбинация битов %s\n",
         itobs(box.us_view,bin_str));
   box.us_view &= ~FILL_MASK;                /* очистить биты фона */
   box.us_view |= (FILL_BLUE | FILL_GREEN);  /* переустановить фон */
   box.us_view ^= OPAQUE;                    /* переключить прозрачность */
   box.us_view |= BORDER_RED;                /* ошибочный подход */
   box.us_view &= ~STYLE_MASK;               /* очистить биты стиля */
   box.us_view |= B_DOTTED;                  /* установить пунктирный стиль */
   printf("\nИзмененные настройки окна:\n");
   show_settings(&box.st_view);
   printf("\nНастройки окна с использованием представления unsigned short:\n");
   show_settings1(box.us_view);
   printf("Комбинация битов %s\n",
         itobs(box.us_view,bin_str));

   return 0;
}
```

```
void show_settings(const struct box_props * pb)
{
   printf("Окно %s.\n",
          pb->opaque == true ? "непрозрачно": "прозрачно");
   printf("Цвет фона %s.\n", colors[pb->fill_color]);
   printf("Рамка %s.\n",
          pb->show_border == true ? "отображается" : "не отображается");
   printf("Цвет рамки %s.\n", colors[pb->border_color]);
   printf("Стиль рамки ");
   switch(pb->border_style)
   {
      case SOLID  : printf("сплошной.\n"); break;
      case DOTTED : printf("пунктирный.\n"); break;
      case DASHED : printf("штриховой.\n"); break;
      default     : printf("неизвестного типа.\n");
   }
}
void show_settings1(unsigned short us)
{
   printf("Окно %s.\n",
         (us & OPAQUE) == OPAQUE? "непрозрачно": "прозрачно");
   printf("Цвет фона %s.\n",
          colors[(us >> 1) & 07]);
   printf("Рамка %s.\n",
         (us & BORDER) == BORDER? "отображается" : "не отображается");
   printf ("Стиль рамки ");
   switch(us & STYLE_MASK)
   {
      case B_SOLID  : printf("сплошной.\n"); break;
      case B_DOTTED : printf("пунктирный.\n"); break;
      case B_DASHED : printf("штриховой.\n"); break;
      default       : printf("неизвестного типа.\n");
   }
   printf("Цвет рамки %s.\n",
         colors[(us >> 9) & 07]);
}
char * itobs(int n, char * ps)
{
   int i;
   const static int size = CHAR_BIT * sizeof(int);
   for (i = size - 1; i >= 0; i--, n >>= 1)
      ps[i] = (01 & n) + '0';
   ps[size] = '\0';
   return ps;
}
```

Ниже приведен вывод.

```
Исходные настройки окна:
Окно непрозрачно.
Цвет фона желтый.
Рамка отображается.
Цвет рамки зеленый.
Стиль рамки штриховой.
```

```
Настройки окна с использованием представления unsigned short:
Окно непрозрачно.
Цвет фона желтый.
Рамка отображается.
Стиль рамки штриховой.
Цвет рамки зеленый.
Комбинация битов 00000000000000000001001010000111

Измененные настройки окна:
Окно прозрачно.
Цвет фона голубой.
Рамка отображается.
Цвет рамки желтый.
Стиль рамки пунктирный.

Настройки окна с использованием представления unsigned short:
Окно прозрачно.
Цвет фона голубой.
Рамка не отображается.
Стиль рамки пунктирный.
Цвет рамки желтый.
Комбинация битов 00000000000000000001011100001100
```

В коде есть несколько моментов, которые необходимо обсудить. Одно из отличий между этими двумя представлениями состоит в том, что побитовому представлению нужна информация о позициях. Например, для представления синего цвета в программе используется константа BLUE, имеющая числовое значение 4. Но поскольку данные организованы в структуре, на самом деле хранить настройку синего цвета для фона будет бит 3 (не забывайте, что нумерация начинается с нуля (см. рис. 15.1)), а настройку синего цвета для рамки – бит 11. Таким образом, в программе определен ряд новых констант:

```
#define FILL_BLUE      0x8
#define BORDER_BLUE    0x800
```

Здесь 0x8 – это значение, когда в 1 установлен только бит 3, а 0x800 – значение, когда в 1 установлен только бит 11. Первую константу можно применять при установке бита синего цвета для фона окна, а вторую – при установке бита синего цвета для рамки. Шестнадцатеричная запись упрощает выяснение, какие биты задействованы. Вспомните, что каждая шестнадцатеричная цифра представляет четыре бита. Следовательно, значение 0x800 соответствует комбинации битов 0x8, но с дописанными восемью битами с состоянием 0. Глядя на десятичные эквиваленты, 2048 и 8, заметить такую связь гораздо труднее.

Если значения являются степенями 2, можно воспользоваться операцией сдвига влево. Например, последние две директивы #define можно заменить следующим образом:

```
#define FILL_BLUE      1<<3
#define BORDER_BLUE    1<<11
```

Во втором операнде указана степень для возведения числа 2. Так, значение 0x8 равно 2^3, а значение 0x800 – 2^{11}. Аналогично, выражение 1<<n дает целочисленное значение, у которого в 1 установлен только бит n. Выражения наподобие 1<<11 являются константными и вычисляются на этапе компиляции.

Вместо директивы #define для создания символических констант можно применять перечисление.

Например, можно поступить так:

```
enum { OPAQUE = 0x1, FILL_BLUE = 0x8, FILL_GREEN = 0x4, FILL_RED = 0x2,
       FILL_MASK = 0xE, BORDER = 0x100, BORDER_BLUE = 0x800,
       BORDER_GREEN = 0x400, BORDER_RED = 0x200, BORDER_MASK = 0xE00,
       B_DOTTED = 0x1000, B_DASHED = 0x2000, STYLE_MASK = 0x3000 };
```

Если вы не намерены создавать переменные с типом этого перечисления, указывать имя в объявлении не обязательно.

Обратите внимание, что использовать побитовые операции для изменения настроек сложнее. В качестве примера давайте установим голубой цвет для фона окна. В данном случае недостаточно просто включить биты, соответствующие синему и зеленому цветам:

```
box.us_view |= (FILL_BLUE | FILL_GREEN);     /* сбросить фон */
```

Дело в том, что цвет также полагается на настройку бита, отвечающего за красный цвет. Если этот бит был включен ранее (скажем, для получения желтого цвета), то приведенный код оставит бит красного цвета установленным и установит в 1 биты синего и зеленого цветов, давая в результате белый цвет. Обойти эту проблему проще всего, сначала отключив все биты, отвечающие за цвет, и лишь затем устанавливать новые значения. Именно поэтому в программе содержится следующий код:

```
box.us_view &= ~FILL_MASK;                   /* очистить биты фона */
box.us_view |= (FILL_BLUE | FILL_GREEN);     /* переустановить фон */
```

Для демонстрации того, что может произойти, если предварительно не очистить соответствующие биты, в программе также предусмотрена такая строка:

```
box.us_view |= BORDER_RED;                   /* ошибочный подход */
```

Из-за того, что бит BORDER_GREEN уже был установлен, результирующим цветом будет BORDER_GREEN | BORDER_RED, что соответствует желтому цвету.

В ситуациях подобного рода применять битовые поля проще:

```
box.st_view.fill_color = CYAN;               /* эквивалент с битовым полем */
```

Тогда нет нужды в предварительной очистке битов. Кроме того, члены битовых полей допускают использование одних и тех же значений цвета для рамки и фона окна. В случае подхода с побитовыми операциями придется применять отличающиеся значения (значения, отражающие действительные позиции битов).

Далее, сравните следующие два оператора вывода:

```
printf("Цвет рамки %s.\n", colors[pb->border_color]);
printf("Цвет рамки %s.\n", colors[(us >> 9) & 07]);
```

В первом операторе выражение pb->border_color имеет значение из диапазона 0–7, поэтому его можно использовать как индекс в массиве colors. Получить ту же информацию с помощью побитовых операций сложнее. Один из подходов предусматривает применение ui >> 9 для сдвига битов цвета рамки в самую правую позицию (биты 0–2) с последующим объединением полученного значения с маской 07, в результате чего все биты кроме трех самых правых будут отключены. То, что осталось, будет находиться в диапазоне 0–7 и может использоваться в качестве индекса для массива colors.

Внимание!

Соответствие между битовыми полями и позициями битов зависит от реализации. Например, при выполнении программы из листинга 15.4 в старой системе Macintosh PowerPC получается следующий вывод:

```
Исходные настройки окна:
Окно непрозрачно.
Цвет фона желтый.
Рамка отображается.
Цвет рамки зеленый.
Стиль рамки штриховой.

Настройки окна с использованием представления unsigned short:
Окно прозрачно.
Цвет фона черный.
Рамка не отображается.
Стиль рамки сплошной.
Цвет рамки черный.
Комбинация битов 10110000101010000000000000000000

Измененные настройки окна:
Окно прозрачно.
Цвет фона желтый.
Рамка отображается.
Цвет рамки зеленый.
Стиль рамки штриховой.

Настройки окна с использованием представления unsigned short:
Окно прозрачно.
Цвет фона голубой.
Рамка отображается.
Стиль рамки пунктирный.
Цвет рамки красный.
Комбинация битов 10110000101010000001001000001101
```

Здесь изменения затрагивают те же биты, что и ранее, но в системе Macintosh PowerPC загрузка структур в память осуществляется по-другому. В частности, первое битовое поле загружается, начиная со старшего, а не младшего бита. Поэтому представление структуры касается первых 16 битов (которые следуют в порядке, отличающемся от версии IBM PC), тогда как представление `unsigned int` затрагивает последние 16 битов. Таким образом, допущения, сделанные относительно позиций битов в листинге 15.4, для Macintosh PowerPC некорректны, а побитовые операции, применяемые для изменения настроек прозрачности и цвета фона, изменяют не те биты.

Средства выравнивания (C11)

Средства выравнивания C11 по своей природе больше ориентированы на манипулирование байтами, чем битами, но они также отражают возможность языка C иметь дело с оборудованием. В этом контексте выравнивание относится к тому, как объекты располагаются в памяти. Например, для максимальной эффективности система может требовать, чтобы значение типа `double` хранилось в памяти по адресу, кратному 4, но разрешать значению типа `char` храниться по любому адресу. Большинству программистов редко когда придется заботиться о выравнивании. Но в некоторых ситуациях контроль над выравниванием позволяет извлечь выгоду, например, при передаче данных из одного физического места в другое либо при вызове инструкций, которые оперируют на множестве элементов данных одновременно.

Операция _Alignof выдает требования к выравниванию указанного типа. Для ее использования необходимо после ключевого слова _Alignof поместить имя типа в круглых скобках:

```
size_t d_align = _Alignof(float);
```

Полученное значение, скажем, 4 для d_align, говорит о том, что объекты float имеют требование к выравниванию, соответствующее 4. Это означает, что 4 является количеством байтов между следующими друг за другом адресами для хранения значений упомянутого типа. В общем случае значения выравнивания должны быть неотрицательными целыми числами, которые представляют собой степень 2. Более высокие значения выравнивания считаются *более жесткими* или *более строгими*, чем меньшие значения, в то время как меньшие значения трактуются как *более слабые*.

С помощью спецификатора _Alignas можно запрашивать конкретное выравнивание для переменной или типа. Однако вы не должны запрашивать выравнивание, которое слабее фундаментального выравнивания, принятого для типа. Например, если требование к выравниванию для float составляет 4, не запрашивайте значение выравнивания, равное 1 или 2. Этот спецификатор применяется как часть объявления, и за ним следует пара круглых скобок, содержащая либо значение выравнивания, либо тип:

```
_Alignas(double) char c1;
_Alignas(8) char c2;
unsigned char _Alignas(long double) c_arr[sizeof(long double)];
```

На заметку!

На момент написания книги компилятор Clang (версии 3.2) требовал, чтобы спецификатор _Alignas(*тип*) располагался после спецификатора типа, как в третьей строке приведенного выше кода. Тем не менее, компилятор GCC 4.7.3 распознает оба порядка следования, как и последующая версия (3.3) компилятора Clang.

В листинге 15.5 показан короткий пример использования _Alignas и _Alignof.

Листинг 15.5. Программа align.c

```
// align.c -- использование _Alignof и _Alignas (C11)
#include <stdio.h>
int main(void)
{
   double dx;
   char ca;
   char cx;
   double dz;
   char cb;
   char _Alignas(double) cz;

   printf("Выравнивание char:   %zd\n", _Alignof(char));
   printf("Выравнивание double: %zd\n", _Alignof(double));
   printf("&dx: %p\n", &dx);
   printf("&ca: %p\n", &ca);
   printf("&cx: %p\n", &cx);
   printf("&dz: %p\n", &dz);
   printf("&cb: %p\n", &cb);
   printf("&cz: %p\n", &cz);

   return 0;
}
```

Вот пример вывода:

```
Выравнивание char:   1
Выравнивание double: 8
&dx: 0x7fff5fbff660
&ca: 0x7fff5fbff65f
&cx: 0x7fff5fbff65e
&dz: 0x7fff5fbff650
&cb: 0x7fff5fbff64f
&cz: 0x7fff5fbff648
```

В нашей системе значение выравнивания 8 для типа double подразумевает, что значения этого типа сохраняются по адресам, кратным 8. Шестнадцатеричные адреса, заканчивающиеся на 0 или 8, являются кратными 8, и адреса такого вида применялись для двух переменных double, а также переменной char по имени cz, которой было назначено значение выравнивания для типа double. Поскольку значением выравнивания для char было 1, компилятор мог использовать для переменных этого типа любые адреса.

Включение заголовочного файла stdalign.h позволяет применять псевдонимы alignas и alignof для _Alignas и _Alignof. Они соответствуют ключевым словам в C++. В C11 также появилась возможность выравнивания для выделенной памяти за счет добавления в библиотеку stdlib.h новой функции распределения памяти со следующим прототипом:

```
void *aligned_alloc(size_t alignment, size_t size);
```

В первом параметре указывается требуемое выравнивание, а во втором — количество необходимых байтов, которое должно быть кратным значению первого параметра. Как и в случае других функций распределения памяти, по завершении работы с выделенной памятью используйте функцию free(), чтобы ее освободить.

Ключевые понятия

Одной из особенностей, которая отличает C от большинства языков высокого уровня, является возможность доступа к отдельным битам целого числа. Это часто играет ключевую роль при взаимодействии с аппаратными устройствами и операционными системами.

Язык C обладает двумя основными средствами для работы с битами. Первое — это семейство побитовых операций, а второе — создание битовых полей в структуре.

В C11 добавлена возможность контроля требования к выравниванию памяти и запрашивания более строгих таких требований.

Обычно, хотя и не всегда, программы, в которых задействованы эти средства, привязаны к конкретным аппаратным платформам или операционным системам и не задуманы быть переносимыми.

Резюме

Вычислительные системы тесно связаны с двоичной системой счисления, поскольку нули и единицы могут представлять выключенное и включенное состояние битов памяти и регистров. Хотя язык C не разрешает записывать целые числа в двоичной форме, он распознает восьмеричные и шестнадцатеричные системы записи. Подобно тому, как двоичная цифра представляет один бит, восьмеричная цифра представляет три бита, а шестнадцатеричная — четыре бита. Такая взаимосвязь позволяет сравнительно просто преобразовывать двоичные числа в восьмеричную или шестнадцатеричную форму.

В языке C предлагается несколько побитовых операций, которые так называются из-за того, что воздействуют на каждый бит значения независимым образом. Побитовая операция отрицания (~) инвертирует каждый бит своего операнда, преобразуя 1 в 0 и наоборот. Побитовая операция "И" (&) формирует значение из двух операндов. Каждый бит в значении устанавливается в 1, если соответствующие биты в обоих операндах равны 1; в противном случае бит устанавливается в 0. Побитовая операция "ИЛИ" (|) также формирует значение из двух операндов. Каждый бит в значении устанавливается в 1, если соответствующий бит в одном из двух или в обоих операндах равен 1; в противном случае бит устанавливается в 0. Побитовая операция исключающего "ИЛИ" (^) действует аналогично с тем отличием, что результирующий бит устанавливается в 1, только если соответствующий бит равен 1 в одном или в другом операнде, но не в обоих.

Вдобавок в C имеются операции сдвига влево (<<) и вправо (>>). Каждая из них создает значение, формируемое путем сдвига битов (влево или вправо) левого операнда на количество позиций, указанное в правом операнде. При операции сдвига влево освобождаемые биты устанавливаются в 0. При операции сдвига вправо освобождаемые биты устанавливаются в 0 для значений без знака. Для значений со знаком поведение операции сдвига вправо зависит от реализации.

Для обращения к отдельным битам или к группе битов в значении можно применять битовые поля. Детали такого манипулирования зависят от реализации.

С помощью операции `_Alignas` можно устанавливать требования к выравниванию при сохранении данных. Инструменты для работы с битами помогают программам на C взаимодействовать с оборудованием, поэтому они чаще всего привязаны к контексту конкретной реализации.

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля приведены в приложении А.

1. Преобразуйте следующие десятичные значения в двоичную форму:
   а. 3
   б. 13
   в. 59
   г. 119
2. Преобразуйте следующие двоичные значения в десятичную, восьмеричную и шестнадцатеричную форму:
   а. `00010101`
   б. `01010101`
   в. `01001100`
   г. `10011101`
3. Вычислите следующие выражения; предположите, что каждое значение имеет 8 битов:
   а. `~3`
   б. `3 & 6`
   в. `3 | 6`
   г. `1 | 6`
   д. `3 ^ 6`
   е. `7 >> 1`
   ж. `7 << 2`

4. Вычислите следующие выражения; предположите, что каждое значение имеет 8 битов:
   а. `~0`
   б. `!0`
   в. `2 & 4`
   г. `2 && 4`
   д. `2 | 4`
   е. `2 || 4`
   ж. `5 << 3`
5. Поскольку в ASCII-коде используются только последние 7 битов, иногда желательно маскировать остальные биты. Как будет выглядеть подходящая маска в двоичной форме? В десятичной? В восьмеричной? В шестнадцатеричной?
6. В листинге 15.2 следующий код

```
while (bits-- > 0)
{
   mask |= bitval;
   bitval <<= 1;
}
```

   можно заменить таким фрагментом:

```
while (bits-- > 0)
{
   mask += bitval;
   bitval *= 2;
}
```

   и программа по-прежнему будет работать. Означает ли это, что действие `*= 2` эквивалентно `<<= 1`? А как насчет `|=` и `+=`?
7. а. Компьютер Tinkerbell содержит в специальном байте информацию, касающуюся оборудования. Этот байт может быть прочитан программой, и он содержит следующую информацию:

Биты	Описание
0–1	Количество дисководов 1,44 Мбайт
2	Не используется
3–4	Количество приводов чтения компакт-дисков
5	Не используется
6–7	Количество жестких дисков

   Подобно IBM PC, компьютер Tinkerbell заполняет битовые поля структуры справа налево. Создайте шаблон битовых полей, подходящий для хранения информации.

   б. Компьютер Klinkerbell, ближайший клон Tinkerbell, заполняет битовые поля структур слева направо. Создайте соответствующий шаблон битовых полей для системы Klinkerbell.

Упражнения по программированию

1. Напишите функцию, которая преобразует строку с двоичным представлением в числовое значение. Другими словами, если есть

   ```
   char * pbin = "01001001";
   ```

   то переменную pbin можно передать этой функции в качестве аргумента, и функция должна возвратить значение 25 типа int.

2. Напишите программу, которая читает две строки с двоичным представлением как аргументы командной строки и выводит результаты применения операции ~ к каждому числу, а также результаты применения операций &, | и ^ к паре чисел. Отобразите результаты в виде двоичных строк. (Если среда командной строки недоступна, обеспечьте в программе интерактивный ввод строк.)

3. Напишите функцию, которая принимает аргумент типа int и возвращает количество включенных битов в нем. Протестируйте функцию в какой-нибудь программе.

4. Напишите функцию, которая принимает два аргумента типа int: значение и позицию бита. Функция должна возвращать 1, если бит в этой позиции равен 1, и 0 в противном случае. Протестируйте функцию в какой-нибудь программе.

5. Напишите функцию, которая циклически сдвигает биты значения типа unsigned int на указанное количество позиций влево. Например, функция rotate_l(x,4) перемещает биты значения x на четыре позиции влево, при этом утраченные слева биты воспроизводятся в правой части комбинации. Другими словами, вытесненный старший бит помещается в позицию младшего бита. Протестируйте функцию в какой-нибудь программе.

6. Разработайте структуру битовых полей, которая содержит следующую информацию:

   - Идентификатор шрифта: число от 0 до 255
   - Размер шрифта: число от 0 до 127
   - Выравнивание: число от 0 до 2, представляющее опции выравнивания влево, по центру и вправо
   - Полужирный: отключен (0) или включен (1)
   - Курсив: отключен (0) или включен (1)
   - Подчеркнутый: отключен (0) или включен (1)

   Используйте эту структуру в программе, которая отображает параметры шрифта и дает пользователю возможность менять параметры с помощью циклического меню. Ниже приводится пример выполнения программы:

   ```
   ИД   РАЗМЕР   ВЫРАВНИВАНИЕ      Ж       К       Ч
    1     12       влево         откл.   откл.   откл.

   ш) изменить шрифт   р) изменить размер    в) изменить выравнивание
   ж) полужирный       к) курсив             п) подчеркнутый
   з) завершить
   р
   ```

```
Введите размер шрифта (0-127): 36
ИД   РАЗМЕР   ВЫРАВНИВАНИЕ      Ж      К      Ч
 1      36     влево         откл.  откл.  откл.

ш) изменить шрифт   р) изменить размер    в) изменить выравнивание
ж) полужирный       к) курсив             п) подчеркнутый
з) завершить
в
Выберите выравнивание:
л) влево            ц) по центру          п) вправо
п

ИД   РАЗМЕР   ВЫРАВНИВАНИЕ      Ж      К      Ч
 1      36     вправо        откл.  откл.  откл.

ш) изменить шрифт   р) изменить размер    в) изменить выравнивание
ж) полужирный       к) курсив             п) подчеркнутый
з) завершить
к

ИД   РАЗМЕР   ВЫРАВНИВАНИЕ      Ж      К      Ч
 1      36     вправо        откл.  вкл.   откл.

ш) изменить шрифт   р) изменить размер    в) изменить выравнивание
ж) полужирный       к) курсив             п) подчеркнутый
з) завершить
з
Программа завершена.
```

Чтобы обеспечить преобразование вводимых значений идентификатора и размера шрифта в значения из указанного диапазона, программа должна применять операцию `&` и подходящие маски.

7. Напишите программу с таким же поведением, как в упражнении 6, но используйте для хранения информации о шрифте переменную типа `unsigned long`, а для манипулирования этой информацией — побитовые операции вместо членов структуры с битовыми полями.

16

Препроцессор и библиотека C

В ЭТОЙ ГЛАВЕ...

- Директивы препроцессора: #define, #include, #ifdef, #else, tendif, #ifndef, #if, #elif, #line, #error, #pragma
- Ключевые слова: _Generic, _Noreturn, _Static_assert
- Функции/макросы:sqrt(), atan(), atan2(), exit(), atexit(), assert(), memcpy(), memmove(), va_start(), va_arg(), va_copy(), va_end()
- Дополнительные возможности препроцессора C
- Функциональные макросы и условная компиляция
- Встраиваемые функции
- Библиотека C и ее некоторые удобные функции

Язык С построен на основе ключевых слов, выражений, операторов, а также правил их использования. Однако стандарт С не ограничивается описанием одного лишь языка. В нем также определено, что должен делать препроцессор, установлено, какие функции формируют стандартную библиотеку С, и детализировано, каким образом работают эти функции. В этой главе мы исследуем препроцессор и библиотеку С, и начнем мы с препроцессора.

Препроцессор, согласно своему названию, анализирует программу до ее компиляции. Следуя указанным директивам, препроцессор заменяет символические сокращения в программе сущностями, которые они представляют. По вашему запросу препроцессор может включать другие файлы, и вы можете выбирать, какой код будет видеть компилятор. Препроцессору ничего не известно о языке С. По существу он преобразует один текст в другой. Правда, такое описание не дает точного представления об истинной пользе и значимости препроцессора, поэтому давайте перейдем к примерам. Вы уже неоднократно встречали директивы `#define` и `#include`. Теперь можно объединить и расширить полученные знания.

Первые шаги в трансляции программы

До передачи управления препроцессору компилятор должен провести программу через ряд этапов трансляции. Компилятор начинает свою работу с того, что устанавливает соответствие символов исходного кода с исходным набором символов. При этом обрабатываются многобайтные символы и триграфы – расширения символов, которые обеспечивают интернациональное применение языка С. (Обзор этих расширений приведен в справочном разделе VII приложения Б.)

Во вторую очередь компилятор обнаруживает все вхождения обратной косой черты с последующим символом новой строки и удаляет их. В результате две физические строки, такие как

```
printf("Это было вели\
колепно!\n");
```

преобразуются в одну *логическую строку*:

```
printf("Это было великолепно!\n");
```

Обратите внимание, что в этом контексте "символ новой строки" означает символ, сгенерированный нажатием клавиши <Enter> для перехода на следующую строку в файле исходного кода, а не символическое представление `\n`.

Это необходимо для подготовки к предварительной обработке, поскольку препроцессор требует, чтобы выражения имели длину, равную одной логической строке, но одна логическая строка может распространяться на несколько физических строк.

Далее компилятор разбивает текст на последовательность препроцессорных лексем, а также на последовательности пробельных символов и комментариев. (В базовой терминологии лексемы представляют собой группы, отделяемые друг от друга пробелами, табуляциями или разрывами строк; позже мы рассмотрим лексемы более подробно.) Сейчас интересно отметить, что каждый комментарий заменяется одним символом пробела. Таким образом, код следующего вида:

```
int/* это не похоже на пробел */fox;
```

превращается в

```
int fox;
```

Кроме того, в рамках реализации компилятора может быть принято решение заменять каждую последовательность пробельных символов (кроме символа новой строки) одиночным пробелом. Наконец, программа готова для этапа предварительной обработки, и препроцессор начинает поиск своих потенциальных директив, обозначаемых символом # в начале строки.

Символические константы: #define

Подобно всем директивам препроцессора, директива #define начинается с символа # в начале строки. Стандарт ANSI и последующие стандарты разрешают предварение символа # пробелами или табуляциями, а также наличие пробела между # и остальной частью директивы. Однако в прежних версиях С обычно требовалось, чтобы директива начиналась в крайней левой позиции в строке, а пробелы между символом # и остальной частью директивы не допускались. Директива может находиться в любом месте файла исходного кода, и ее определение распространяется от этого места до конца файла. В наших программах мы интенсивно использовали директивы для определения *символических*, или *именованных*, констант. Однако, как вскоре будет показано, область применения директив этим не ограничивается. В листинге 16.1 продемонстрированы некоторые возможности и свойства директивы #define.

Листинг 16.1. Программа preproc.c

```
/* preproc.c -- простые примеры работы с препроцессором */
#include <stdio.h>
#define TWO 2              /* при желании можно использовать комментарии */
#define OW "Логика - последнее убежище лишенных\
воображения. - Оскар Уайльд"  /* обратная косая черта переносит определение */
                           /* на следующую строку                          */
#define FOUR TWO*TWO
#define PX printf("X = %d.\n", x)
#define FMT "X = %d.\n"
int main(void)
{
   int x = TWO;

   PX;
   x = FOUR;
   printf(FMT, x);
   printf("%s\n", OW);
   printf("TWO: OW\n");

   return 0;
}
```

Директива препроцессора простирается до тех пор, пока не встретится первый символ новой строки после знака #. Другими словами, длина директивы ограничена одной строкой. Однако, как уже упоминалось ранее, комбинации обратной косой черты и символа новой строки удаляются до начала работы препроцессора, поэтому директиву можно распространить на несколько физических строк. Тем не менее, эти строки образуют одну логическую строку.

Каждая строка #define (т.е. логическая строка) состоит из трех частей. Первая часть — это сама директива #define. Вторая часть — выбранное программистом сокращение, называемое *макросом*. Некоторые макросы, как в приведенном выше примере, представляют значения. Они называются *объектными макросами*. (В языке С еще существуют *функциональные макросы*, о которых речь пойдет позже.)

Имя макроса не должно содержать пробелов. На макросы распространяются правила именования переменных: разрешены только буквы, цифры и символ подчеркивания (_), а первым символом не должна быть цифра. Третья часть (остаток строки) называется *списком замены* или *телом* (рис. 16.1). Когда препроцессор обнаруживает в программе имя одного из макросов, он почти всегда заменяет его телом. (Как вскоре будет показано, из этого правила существует одно исключение.) Этот процесс перехода от макроса к подставляемому итоговому значению называется *расширением*. Обратите внимание, что в строке #define могут быть указаны стандартные комментарии; ранее уже говорилось о том, до начала работы препроцессора каждый комментарий заменяется пробелом.

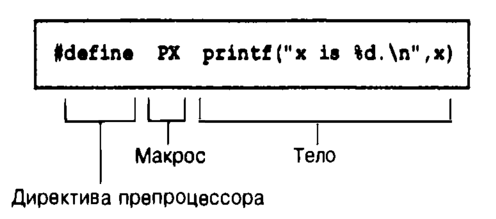


Рис. 16.1. Части определения объектного макроса

Давайте запустим программу и посмотрим, как она работает:

```
X = 2.
X = 4.
Логика - последнее убежище лишенных воображения. - Оскар Уайльд
TWO: OW
```

Здесь вот что происходит. Оператор

```
int x = TWO;
```

преобразуется следующим образом:

```
int x = 2;
```

поскольку вместо TWO было подставлено 2. Затем оператор

```
PX;
```

приобретает следующий вид:

```
printf("X = %d.\n", x);
```

Подстановка осуществилась для всего оператора целиком. Это новый прием, т.к. до сих пор мы использовали макросы только для представления констант. Здесь вы видите, что макрос может представлять любую строку, даже целое выражение С. Однако отметим, что это константная строка; макрос PX будет выводить только значение переменной по имени x.

В следующей строке также демонстрируется новый прием. Может показаться, что FOUR будете заменено 4, но на самом деле процесс был другим. Строка

```
x = FOUR;
```

преобразуется в строку

```
x = TWO*TWO;
```

которая затем становится следующей:

```
x = 2*2;
```

На этом процесс расширения макроса завершен. Действительное умножение происходит не во время работы препроцессора, а на этапе компиляции, поскольку компилятор С на этой стадии вычисляет все константные выражения (т.е. выражения, которые содержат только константы). Препроцессор не выполняет вычислений, а просто совершенно буквально производит указанные с помощью директив подстановки.

Обратите внимание, что определение макроса может включать другие макросы. (Некоторые компиляторы такую вложенность макросов не поддерживают.)

Следующая строка

```
printf (FMT, x);
```

приводится к виду:

```
printf("X = %d.\n",x);
```

потому что FMT заменяется соответствующей строкой. Такой подход может оказаться удобным при наличии длинной управляющей строки, которую приходится применять несколько раз. Вместо этого можно было поступить следующим образом:

```
const char * fmt = "X = %d.\n";
```

Затем можно использовать fmt в качестве управляющей строки для printf().

В следующей строке OW заменяется соответствующей строкой. Двойные кавычки делают строку замещения символьной строковой константой. Компилятор сохранит ее в массиве с завершающим нулевым символом. Таким образом, директива

```
#define HAL 'Z'
```

определяет символьную константу, а директива

```
#define HAP "Z"
```

определяет символьную строку: Z\0.

В этом примере мы применяли обратную косую черту непосредственно перед концом строки, распространяя директиву на следующую строку:

```
#define OW "Логика - последнее убежище лишенных\
воображения. - Оскар Уайльд"
```

Обратите внимание, что вторая строка выровнена влево. Если бы директива имела вид:

```
#define OW "Логика - последнее убежище лишенных\
    воображения. - Оскар Уайльд"
```

то вывод был бы таким:

```
Логика - последнее убежище лишенных    воображения. - Оскар Уайльд
```

Пробелы с начала строки и до слова воображения считаются частью строки. Обычно где бы препроцессор ни обнаружил в программе один из макросов, он заменяет его литерально эквивалентным текстом замены. Если строка замещения содержит вложенные макросы, они также заменяются. Единственным исключением при замене является ситуация, когда найденный макрос заключен в двойные кавычки. Поэтому строка

```
printf("TWO: OW");
```

выводит текст: TWO: OW буквально вместо того, чтобы вывести

```
2: Логика - последнее убежище лишенных воображения. - Оскар Уайльд
```

Для вывода показанной строки понадобится следующий код:

```
printf("%d: %s\n", TWO, OW);
```

Здесь имя макроса находится за пределами двойных кавычек.

Когда должны использоваться символические константы? Вы должны их применять для большинства числовых констант. Если число представляет собой некоторую константу, участвующую в вычислениях, то символическое имя сделает ее назначение более понятным. Если число является размером массива, то символическое имя упростит его будущее изменение и корректировку границ выполнения циклов. Если число представляет собой системный код, такой как EOF, то символическое имя увеличит степень переносимости программы; понадобится изменять только одно определение EOF. Мнемоническое значения, изменяемость и переносимость — все эти характеристики делают символические константы заслуживающими внимания.

Однако ключевое слово const, которое теперь поддерживается в С, обеспечивает более гибкий способ создания констант. С помощью const можно создавать глобальные и локальные константы, числовые константы, константы в форме массивов и константы в виде структур. С другой стороны, константы-макросы могут использоваться для указания размеров стандартных массивов, а также инициализирующих значений для величин const.

```
#define LIMIT 20
const int LIM = 50;
static int data1[LIMIT];      // допустимо
static int data2[LIM];        // не обязательно должно быть допустимым
const int LIM2 = 2 * LIMIT;   // допустимо
const int LIM3 = 2 * LIM;     // не обязательно должно быть допустимым
```

Обратите внимание на комментарий "не обязательно должно быть допустимым". В языке С предполагается, что размер массива для неавтоматических массивов задается целочисленным константным выражением, т.е. комбинацией целочисленных констант наподобие 5, констант из перечислений и выражений sizeof. Значения, объявленные с применением const, сюда не входят. (В этом отношении С отличается от C++, где значения const могут быть частью константных выражений.) Тем не менее, реализация компилятора может адаптировать другие формы константных выражений. В результате, к примеру, GCC 4.7.3 не примет объявление для data2, но Clang 4.6 — примет.

Лексемы

Формально тело макроса должно быть строкой *лексем*, а не строкой символов. Лексемы препроцессора С — это отдельные "слова" в теле определения макроса. Они отделяются друг от друга пробельными символами. Например, определение

```
#define FOUR 2*2
```

имеет одну лексему — последовательность 2*2, но определение

```
#define SIX 2 * 3
```

содержит три лексемы: 2, * и 3.

Строки символов и строки лексем отличаются в том, как трактуются последовательности из множества пробелов. Рассмотрим следующее определение:

```
#define EIGHT 4    *    8
```

Препроцессор, который интерпретирует тело макроса как строку символов, вместо EIGHT подставляет 4   *   8. То есть дополнительные пробелы будут частью замены, но препроцессор, который интерпретирует тело как строку лексем, заменит EIGHT тремя лексемами, разделенными одиночными пробелами: 4 * 8. Другими словами, интерпретация в виде строки символов трактует пробелы как часть тела, а ин-

терпретация в виде строки лексем считает пробелы разделителями между лексемами внутри тела. На практике некоторые компиляторы С рассматривают тела макросов как строки, а не как лексемы. Это различие имеет практическое значение только для более сложных случаев использования по сравнению с приведенными здесь.

Кстати, в компиляторе С принята более сложная трактовка лексем по сравнению с препроцессором. Компилятор понимает правила языка С и не обязательно требует наличия пробелов для отделения лексем друг от друга. Например, компилятор С будет интерпретировать 2*2 как три лексемы, поскольку он выясняет, что 2 является константой, а * — операцией.

Переопределение констант

Предположим, что вы определили константу LIMIT как имеющую значение 20, и затем в том же файле определили ее снова, но уже со значением 25. Такой процесс называется *переопределением константы*. Политика переопределения зависит от реализации компилятора. Одни реализации считают переопределение ошибкой, если только новое определение не совпадает со старым. Другие разрешают переопределение, возможно, выдавая предупреждение. В стандарте ANSI принят первый вариант, разрешающий переопределение, только если новое определение дублирует предыдущее.

Совпадение определений означает, что их тела должны иметь одни и те же лексемы в том же самом порядке. Поэтому приведенные ниже определения эквивалентны:

```
#define SIX 2 * 3
#define SIX 2   *  3
```

Оба определения содержат те же самые три лексемы, а избыточные пробелы не являются частью тела. Следующее определение рассматривается как отличающееся:

```
#define SIX 2*3
```

Оно содержит только одну лексему, а не три, поэтому не совпадает с предыдущими определениями. Если вы хотите переопределить макрос, применяйте директиву #undef, которую мы обсудим позже.

Если требуется переопределить некоторые константы, то для достижения цели может быть проще использовать ключевое слово const и правила области действия.

Использование аргументов в директиве #define

С помощью аргументов можно создавать *функциональные макросы*, которые выглядят и действуют во многом подобно обычным функциям. Макрос с аргументами очень похож на функцию, потому что аргументы заключаются в круглые скобки. Определения функциональных макросов имеют один или более аргументов в скобках, и эти аргументы затем присутствуют в выражении замены, как показано на рис. 16.2.

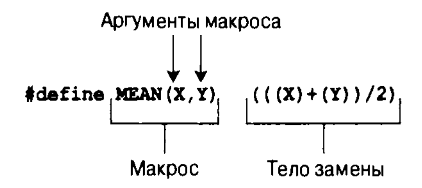


Рис. 16.2. Части определения функционального макроса

Вот пример определения:

```
#define SQUARE(X) X*X
```

Оно может применяться в программе следующим образом:

```
z = SQUARE(2);
```

Оператор выглядит похожим на вызов функции, хотя поведение макроса не обязательно будет идентичным. В листинге 16.2 иллюстрируется использование этого и второго макроса. В некоторых примерах также обращается внимание на возможные ловушки, поэтому читайте их внимательно.

Листинг 16.2. Программа mac_arg.c

```
/* mac_arg.c -- макросы с аргументами */
#include <stdio.h>
#define SQUARE(X) X*X
#define PR(X) printf("Результат: %d.\n", X)
int main(void)
{
    int x = 5;
    int z;

    printf("x = %d\n", x);
    z = SQUARE(x);
    printf("Вычисление SQUARE(x): ");
    PR(z);
    z = SQUARE(2);
    printf("Вычисление SQUARE(2): ");
    PR(z);
    printf("Вычисление SQUARE(x+2): ");
    PR(SQUARE(x+2));
    printf("Вычисление 100/SQUARE(2): ");
    PR(100/SQUARE(2));
    printf("x = %d.\n", x);
    printf("Вычисление SQUARE(++x): ");
    PR(SQUARE(++x));
    printf("После инкрементирования x = %x.\n", x);

    return 0;
}
```

Макрос SQUARE имеет следующее определение:

```
#define SQUARE(X) X*X
```

Здесь SQUARE — идентификатор макроса, X в SQUARE(X) — аргумент макроса, а X*X — список замены. Каждое вхождение SQUARE(x) в листинге 16.2 заменяется x*x. Отличие данного примера от предыдущих состоит в возможности использования в макросе любых символов помимо X. Символ X в определении макроса заменяется символом, который указан при вызове макроса в программе. Таким образом, SQUARE(2) заменяется 2*2, так что X действительно играет роль аргумента.

Однако, как вскоре будет показано, аргумент макроса не работает в точности как аргумент функции. Ниже представлены результаты выполнения программы. Обратите внимание, что некоторые вычисления дают результат, отличающийся от того, что можно было ожидать. На самом деле ваш компилятор может даже выдать не такой результат, как приведенный в предпоследней строке:

```
x = 5
Вычисление SQUARE(x): Результат: 25.
Вычисление SQUARE(2): Результат: 4.
Вычисление SQUARE(x+2): Результат: 17.
Вычисление 100/SQUARE(2): Результат: 100.
x = 5.
Вычисление SQUARE(++x): Результат: 42.
После инкрементирования x = 7.
```

Первые две строки вполне предсказуемы, но затем встречается несколько странных результатов. Вспомните, что x имеет значение 5. Это может привести к предположению, что SQUARE(x+2) должно быть 7*7, или 49, но выводится 17 — простое число, но определенно не квадрат! Причина такого вводящего в заблуждение вывода связана с тем, что, как уже говорилось, препроцессор не выполняет вычислений, а просто заменяет последовательности символов. Где бы в определении не появлялось x, препроцессор подставит вместо x символы x+2. Поэтому

```
x*x
```

принимает вид:

```
x+2*x+2
```

Единственным умножением является 2*x. Если x равно 4, выражение вычисляется следующим образом:

```
4 + 2 * 4 + 2 = 4 + 8 + 2 = 14
```

Этот пример подчеркивает важное отличие между вызовом функции и вызовом макроса. При вызове функции ей передается значение аргумента во время выполнения программы. При вызове макроса лексема аргумента передается в программу перед компиляцией; это другой процесс, происходящий в другое время. Можно ли исправить определение, чтобы вызов SQUARE(x+2) выдавал 36? Конечно. Нужны просто дополнительные круглые скобки:

```
#define SQUARE(x) (x)*(x)
```

Теперь SQUARE(x+2) превращается в (x+2)*(x+2), и вы получите ожидаемое умножение, поскольку круглые скобки останутся в заменяющей строке.

Однако это не решает всех проблем. Рассмотрим события, которые приводят к тому, что следующая строка:

```
100/SQUARE(2)
```

в выводе преобразуется к виду

```
100/2*2
```

Согласно приоритетам операций, выражение вычисляется слева направо: (100/2)*2, или 50*2, или 100. Для устранения путаницы SQUARE(x) необходимо определить так:

```
#define SQUARE(x) (x*x)
```

В результате это дает 100/(2*2), что в итоге вычисляется как 100/4, или 25.

В следующем определении учтены ошибки обоих примеров:

```
#define SQUARE(x) ((x)*(x))
```

Из всего продемонстрированного можно извлечь такой урок: применяйте столько круглых скобок, сколько необходимо для того, чтобы обеспечить корректный порядок выполнения операций.

Но даже эти меры предосторожности не спасают от ошибки в последнем примере:

```
SQUARE(++x)
```

В результате получается:

```
++x*++x
```

Здесь x инкрементируется дважды — один раз до операции умножения и один раз после нее:

```
++x*++x = 6*7 = 42
```

Из-за того, что выбор конкретного порядка выполнения операций оставлен за разработчиками реализаций, некоторые компиляторы генерируют умножение 7*6. Есть компиляторы, которые могут инкрементировать оба операнда перед умножением, выдавая в результате 7*7, или 49. На самом деле вычисление этого выражения приводит к ситуации, которая в стандарте называется неопределенным поведением. Тем не менее, во всех этих случаях x начинает со значения 5 и заканчивает значением 7, хотя код выглядит так, будто инкрементирование происходит только один раз.

Простейшее решение этой проблемы — избегать использования ++x как аргумента макроса. Вообще лучше не применять в макросах операции инкремента и декремента. Следует отметить, что выражение ++x будет работать в качестве аргумента функции, т.к. оно вычисляется как значение 6, которое затем передается функции.

Создание строк из аргументов макроса: операция

Рассмотрим следующий функциональный макрос:

```
#define PSQR(X) printf("Квадрат X равен %d.\n", ((X)*(X)));
```

Предположим, что этот макрос используется следующим образом:

```
PSQR(8);
```

Вот каким будет вывод:

```
Квадрат X равен 64.
```

Обратите внимание, что определение X в строке, заключенной в двойные кавычки, трактуется как обычный текст, а не лексема, которую можно заменить.

Представим, что вы хотите поместить аргумент макроса в строку. Язык C позволяет сделать это. Внутри заменяющей части функционального макроса символ # становится операцией препроцессора, которая преобразует лексемы в строки. Пусть x является параметром макроса, тогда #x — это имя параметра, преобразованное в строку "x". Такой процесс называется *превращением в строку* и демонстрируется в листинге 16.3.

Листинг 16.3. Программа subst.c

```
/* subst.c -- подстановка в строке */
#include <stdio.h>
#define PSQR(x) printf("Квадрат " #x " равен %d.\n",((x)*(x)))
int main(void)
{
    int y = 5;

    PSQR(y);
    PSQR(2 + 4);

    return 0;
}
```

Вывод выглядит следующим образом:

```
Квадрат y равен 25.
Квадрат 2 + 4 равен 36.
```

В первом вызове макроса #x заменяется строкой "y", а во втором вызове вместо #x подставляется "2 + 4". Конкатенация строк ANSI С затем объединяет эти строки с другими строками в операторе printf() для получения финальной строки. Например, первый вызов макроса дает следующий оператор:

```
printf("Квадрат " "y" " равен %d.\n",((y)*(y)));
```

После этого конкатенация объединяет три расположенные рядом строки в одну:

```
"Квадрат y равен %d.\n"
```

Средство слияния препроцессора: операция

Подобно #, операция ## может применяться в заменяющей части функционального макроса. Вдобавок она может использоваться в заменяющей части объектного макроса. Операция ## объединяет две лексемы в одну. Предположим, вы могли бы записать такое определение:

```
#define XNAME(n) x ## n
```

Тогда макрос

```
XNAME(4)
```

будет расширен следующим образом:

```
x4
```

В листинге 16.4 этот и еще один макрос применяются для слияния лексем с помощью операции ##.

Листинг 16.4. Программа glue.c

```
// glue.c -- использование операции ##
#include <stdio.h>
#define XNAME(n) x ## n
#define PRINT_XN(n) printf("x" #n " = %d\n", x ## n);

int main(void)
{
    int XNAME(1) = 14;     // превращается в int x1 = 14;
    int XNAME(2) = 20;     // превращается в int x2 = 20;
    int x3 = 30;

    PRINT_XN(1);           // превращается в printf("x1 = %d\n", x1);
    PRINT_XN(2);           // превращается в printf("x2 = %d\n", x2);
    PRINT_XN(3);           // превращается в printf("x3 = %d\n", x3);

    return 0;
}
```

Ниже показан вывод:

```
x1 = 14
x2 = 20
x3 = 30
```

Обратите внимание, что в макросе PRINT_XN() операция # используется для объединения строк, а операция ## — для объединения лексем в новый идентификатор.

Макросы с переменным числом аргументов: ... и __VA_ARGS__

Некоторые функции, скажем, printf(), принимают переменное количество аргументов. Обсуждаемый ранее заголовочный файл stdvar.h предоставляет инструменты для создания определяемых пользователем функций с переменным числом аргументов. В C99/C11 то же самое сделано и для макросов.

Идея заключается в том, что последний аргумент в списке аргументов для определения макроса может быть троеточием. Если это так, то в заменяющей части может применяться предопределенный макрос __VA_ARGS__, который будет подставлен вместо троеточия. Для примера рассмотрим следующее определение:

```
#define PR(...) printf(__VA_ARGS__)
```

Предположим, что в программе содержатся вызовы макроса вроде показанных ниже:

```
PR("Здравствуйте");
PR("вес = %d, доставка = $%.2f\n", wt, sp);
```

Для первого вызова __VA_ARGS__ расширяется в один аргумент:

```
"Здравствуйте"
```

Для второго вызова он расширяется в три аргумента:

```
"вес = %d, доставка = $%.2f\n", wt, sp
```

Таким образом, результирующий код выглядит так:

```
printf("Здравствуйте");
printf("вес = %d, доставка = $%.2f\n", wt, sp);
```

В листинге 16.5 приведен более сложный пример, в котором используются конкатенация строк и операция #.

Листинг 16.5. Программа variadic.c

```
// variadic.c -- макросы с переменным числом аргументов
#include <stdio.h>
#include <math.h>
#define PR(X, ...) printf("Сообщение " #X ": " __VA_ARGS__)
int main(void)
{
   double x = 48;
   double y;

   y = sqrt(x);
   PR(1, "x = %g\n", x);
   PR(2, "x = %.2f, y = %.4f\n", x, y);

   return 0;
}
```

В первом вызове макроса X имеет значение 1, так что #X становится "1". В результате получается следующее расширение:

```
print("Сообщение " "1" ": " "x = %g\n", x);
```

Затем осуществляется конкатенация четырех строк, сокращая вызов к такому виду:

```
print("Сообщение 1: x = %g\n", x);
```

В итоге имеем показанный ниже вывод:

```
Сообщение 1: x = 48
Сообщение 2: x = 48.00, y = 6.9282
```

Не забывайте, что троеточие должно быть последним аргументом макроса; следующее определение является ошибочным:

```
#define WRONG(X, ..., Y) #X #__VA_ARGS__ #y   // не работает
```

Выбор между макросом и функцией

Многие задачи могут быть решены за счет применения макроса с аргументами либо функции. Что должно использоваться? Здесь нет каких-то строго определенных правил, но есть ряд соображений, которые следует принимать во внимание.

Макросы несколько сложнее в применении, чем обычные функции, т.к. макросы могут иметь неожиданные побочные эффекты, если вы проявите неосмотрительность. Некоторые компиляторы ограничивают определение макроса одной строкой, и вероятно лучше придерживаться этого ограничения, даже если в вашем компиляторе оно отсутствует.

Выбор между макросом и функцией связан с достижением компромисса между быстродействием и размером кода. Макрос генерирует встраиваемый код, т.е. в программу помещается оператор. Если макрос используется 20 раз, в программу вставляется 20 строк кода. Когда 20 раз применяется функция, в программе все равно содержится только одна копия ее операторов, что уменьшает размер кода. С другой стороны, поток управления программы должен переходить туда, где находится функция, и затем возвращаться в место ее вызова. Этот процесс отнимает больше времени, чем выполнение встраиваемого кода.

Преимущество макросов в том, что они не заботятся о типах переменных. (Причина связана с тем, что они имеют дело со строками символов, а не действительными значениями.) Таким образом, макрос `SQUARE(x)` может с одинаковым успехом использоваться с типом `int` или `float`.

В C99 появилась третья альтернатива — встраиваемые функции. Мы обсудим их позже в этой главе. Программисты обычно применяют макросы для простых функций, таких как перечисленные ниже:

```
#define MAX(X,Y)    ((X) > (Y) ? (X) : (Y))
#define ABS(X)      ((X) < 0 ? -(X) : (X))
#define ISSIGN(X)   ((X) == '+' || (X) == '-' ? 1 : 0)
```

(Последний макрос имеет значение 1, или истинное, если x является символом алгебраического знака.)

Далее указано несколько моментов, о которых не следует забывать.

- Помните, что имя макроса не должно содержать пробелов, но пробелы допускаются в замещающей строке. В ANSI С разрешены пробелы в списке аргументов.
- Заключайте в скобки каждый аргумент и определение в целом. Это гарантирует корректное группирование элементов в выражении следующего рода:

  ```
  forks = 2 * MAX(guests + 3, last);
  ```

- Используйте прописные буквы для имен функциональных макросов. Данное соглашение не так широко распространено, как применение прописных букв в именах константных макросов. Тем не менее, одна из веских причин использования прописных букв связана с тем, что это напоминает вам о возможных побочных эффектах макросов.

- Если вы намерены применять макрос вместо функции главным образом для ускорения работы программы, сначала попытайтесь выяснить, обеспечит ли это заметный выигрыш. Макрос, который используется в программе один раз, не приведет к значительному улучшению скорости ее выполнения. Макрос, находящийся внутри вложенного цикла, является намного лучшим кандидатом для ускорения работы программы. Многие системы предлагают профилировщики программ, которые помогают выявлять фрагменты кода, требующие наибольшего времени выполнения.

Предположим, что вы разработали несколько нужных вам функциональных макросов. Должны ли вы набирать их каждый раз, когда пишется новая программа? Нет, если вы будете помнить о директиве `#include`, которая рассматривается в следующем разделе.

Включение файлов: директива #include

Когда препроцессор встречает директиву `#include`, он ищет файл с указанным в директиве именем и включает его содержимое в текущий файл. Директива `#include` в файле исходного кода заменяется текстом включаемого файла. Это аналогично вводу содержимого включаемого файла в той же позиции внутри исходного файла. Существуют две разновидности `#include`:

```
#include <stdio.h>         ← Имя файла указано в угловых скобках
#include "mystuff.h"       ← Имя файла указано в двойных кавычках
```

В системе Unix угловые скобки сообщают препроцессору о необходимости поиска файла в одном или большем числе стандартных системных каталогов. Двойные кавычки говорят о том, что сначала следует просмотреть текущий каталог (или другой каталог, который указан вместе с именем файла), а затем искать в стандартных каталогах:

```
#include <stdio.h>         ← Поиск в системных каталогах
#include "hot.h"           ← Поиск в текущем рабочем каталоге
#include "/usr/biff/p.h"   ← Поиск в каталоге /usr/biff
```

Интегрированные среды разработки (IDE – Integrated Development Environment) также имеют стандартное местоположение или несколько таких местоположений для системных заголовочных файлов. Многие IDE-среды предоставляют опции меню для указания дополнительных местоположений, которые должны просматриваться в случае применения угловых скобок. Как и в Unix, использование двойных кавычек означает поиск сначала в локальном каталоге, но что это в точности за каталог – зависит от компилятора. Некоторые компиляторы ищут в том же каталоге, где находится исходный код, другие – в текущем рабочем каталоге, а третьи – в каталоге, содержащем файл проекта.

В ANSI C не требуется строгое соблюдение модели каталогов для файлов, т.к. не все вычислительные системы организованы одинаково. Вообще говоря, метод, применяемый для именования файлов, зависит от системы, но использование угловых скобок и двойных кавычек – нет.

Зачем включать файлы? Причина в том, что они содержат информацию, которая необходима компилятору. Например, файл `stdio.h` обычно содержит определения `EOF`, `NULL`, `getchar()` и `putchar()`. Два последних определены как функциональные макросы. Он также содержит прототипы функций ввода-вывода C.

Суффикс .h традиционно применяется для *заголовочных файлов* – файлов с информацией, которая помещается в начале программы. Заголовочные файлы часто содержат операторы препроцессора. Некоторые из них, например, stdio.h, предоставляются системой, но вы можете создавать собственные заголовочные файлы.

Включение крупного заголовочного файла не обязательно приводит к значительному увеличению размера программы. Содержимое заголовочных файлов по большей части является информацией, которая используется компилятором для генерации окончательного кода, а не материалом, добавляемым к этому коду.

Пример заголовочного файла

Предположим, вы разработали структуру для хранения имени и фамилии лица, а также написали функции для работы с этой структурой. Всевозможные объявления вы могли бы собрать вместе внутри заголовочного файла. В листинге 16.6 приведен пример такого файла.

Листинг 16.6. Заголовочный файл names_st.h

```
// names_st.h -- заголовочный файл для структуры names_st
// константы
#include <string.h>
#define SLEN 32

// объявления структур
struct names_st
{
   char first[SLEN];
   char last[SLEN];
};

// определения типов
typedef struct names_st names;

// прототипы функций
void get_names(names *);
void show_names(const names *);
char * s_gets(char * st, int n);
```

Этот заголовочный файл содержит множество типичных для таких файлов элементов: директивы #define, объявления структур, операторы typedef и прототипы функций. Обратите внимание, что ни один из этих элементов не является исполняемым кодом; они представляют собой информацию, применяемую компилятором при создании исполняемого кода.

Показанный заголовочный файл довольно прямолинеен. Обычно вы должны использовать #ifndef и #define, чтобы защититься от многократных включений заголовочного файла. Мы возвратимся к этому приему позже.

Исполняемый код обычно размещается в файле исходного кода, а не в заголовочном файле. Например, в листинге 16.7 показаны определения функций, соответствующие прототипам функций из заголовочного файла. В нем включается заголовочный файл, поэтому компилятору будет знать о типе names_st.

Листинг 16.7. Исходный файл name_st.c

```
// names_st.c -- определение функций для names_st
#include <stdio.h>
#include "names_st.h"    // включение заголовочного файла
```

```
// определения функций
void get_names(names * pn)
{
   printf("Введите свое имя: ");
   s_gets(pn->first, SLEN);

   printf("Введите свою фамилию: ");
   s_gets(pn->last, SLEN);
}

void show_names(const names * pn)
{
   printf("%s %s", pn->first, pn->last);
}

char * s_gets(char * st, int n)
{
   char * ret_val;
   char * find;

   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      find = strchr(st, '\n');      // поиск символа новой строки
      if (find)                     // если адрес является ненулевым,
         *find = '\0';              // поместить туда нулевой символ
      else
         while (getchar() != '\n')
            continue;               // отбросить остальную часть строки
   }
   return ret_val;
}
```

В функции `get_names()` применяется `fgets()` (через `s_gets()`), чтобы избежать переполнения целевых массивов. В листинге 16.8 приведен пример программы, в которой используются показанные ранее заголовочный файл и файл исходного кода.

Листинг 16.8. Программа `useheader.c`

```
// useheader.c -- использование структуры names_st
#include <stdio.h>
#include "names_st.h"
// не забудьте скомпоновать с names_st.c

int main(void)
{
   names candidate;

   get_names(&candidate);
   printf("Добро пожаловать в программу, ");
   show_names(&candidate);
   printf("!\n");

   return 0;
}
```

Вот результаты пробного запуска:

```
Введите свое имя: Иван
Введите свою фамилию: Иванов
Добро пожаловать в программу, Иван Иванов!
```

Обратите внимание на следующие аспекты программы.

- В обоих файлах исходного кода применяется структура names_st, поэтому они оба должны включать заголовочный файл names_st.h.
- Необходимо компилировать и компоновать файлы исходного кода names_st.c и useheader.c.
- Объявления и другие элементы подобного рода содержатся в заголовочном файле names_st.h; определения функций размещены в файле исходного кода names_st.c.

Случаи применения заголовочных файлов

Просмотр содержимого стандартных заголовочных файлов поможет получить представление о том, какого рода информация в них находится. Ниже перечислено наиболее распространенное содержимое этих файлов.

- **Символические константы.** В типичном файле stdio.h, к примеру, определены константы EOF, NULL и BUFSIZ (размер стандартного буфера ввода-вывода).
- **Функциональные макросы.** Например, функция getchar() обычно определена как getc(stdin), а getc() — в форме довольно сложного макроса. Заголовочный файл ctype.h, как правило, содержит определения макросов для функций ctype.
- **Объявления функций.** Заголовочный файл string.h (strings.h в некоторых более старых системах), например, содержит объявления для семейства функций обработки строк. Согласно ANSI С и последующим стандартам, эти объявления представлены в виде прототипов функций.
- **Определения шаблонов структур.** Стандартные функции ввода-вывода используют структуру FILE, содержащую информацию о файле и связанном с ним буфере. Объявление этой структуры находится в файле stdio.h.
- **Определения типов.** Вы можете вспомнить, что стандартные функции ввода-вывода применяют аргумент типа указателя на FILE. Обычно в файле stdio.h используется #define или typedef для того, чтобы имя FILE представляло указатель на структуру. Аналогично, в заголовочных файлах определены типы size_t и time_t.

Многие программисты разрабатывают собственные стандартные заголовочные файлы для применения в своих программах. Это особенно полезно в ситуации, когда вы создаете семейство взаимосвязанных функций и/или структур.

Кроме того, заголовочные файлы можно применять для объявления внешних переменных, с которыми совместно работают несколько файлов. Это целесообразно, например, при разработке семейства функций, совместно использующих переменную для сообщения о некотором состоянии, таком как условие ошибки. В таком случае можно определить переменную с внешним связыванием и областью действия на уровне файла исходного кода, который содержит объявления функций:

```
int status = 0;         // переменная с областью действия
                        // на уровне файла исходного кода
```

Затем в заголовочный файл, связанный с файлом исходного кода, можно поместить ссылочное объявление:

```
extern int status;      // в заголовочном файле
```

Этот код затем появляется в любом файле, в котором был включен данный заголовочный файл, делая переменную доступной файлам, работающим с упомянутым семейством функций. Кроме того, посредством включения это объявление обнаруживается в файле исходного кода функций, однако одном файле допускается наличие определяющего и ссылочного объявлений, если они согласованы по типу.

Еще одним кандидатом для включения в заголовочный является переменная или массив с областью действия на уровне файла, внутренним связыванием и квалификатором const. Часть const предотвращает случайные изменения, а часть static означает, что каждый файл, включающий этот заголовок, получает собственную копию констант. Это устраняет необходимость в наличии одного файла с определяющим объявлением и остальных файлов со ссылочными объявлениями.

Директивы #include и #define являются наиболее интенсивно применяемыми средствами препроцессора С. Остальные директивы будут рассматриваться менее детально.

Другие директивы

Программистам часто приходится создавать программы и библиотечные пакеты на языке С, которые должны работать в разнообразных средах. Виды кода могут варьироваться от среды к среде. Препроцессор предлагает несколько директив, помогающих программисту создавать код, который может переноситься из одной системы в другую за счет изменения значений макросов #define. Директива #undef отменяет предыдущее определение #define. Директивы #if, #ifdef, #ifndef, #else, #elif и #endif позволяют указывать различные варианты кода, подлежащего компиляции. Директива #line дает возможность переустанавливать информацию о строках и файлах, директива #error служит для вывода сообщений об ошибках, а с помощью директивы #pragma можно предоставлять инструкции компилятору.

Директива #undef

Директива #undef отменяет заданное определение #define. Предположим, что есть следующее определение:

```
#define LIMIT 400
```

Тогда директива

```
#undef LIMIT
```

удалит это определение. Затем LIMIT можно переопределить, назначив новое значение. Отмена определения LIMIT допустима даже в случае, если предварительное определение не делалось. Если вы хотите использовать некоторое имя, но не уверены в том, что оно не было определено ранее, на всякий случай его определение можно отменить.

Определение с точки зрения препроцессора

В отношении того, что считать идентификатором, препроцессор следует таким же правилам, как и язык С: идентификатор может состоять только из букв верхнего и нижнего регистра, цифр и символа подчеркивания, а первым символом не может быть цифра. Когда препроцессор встречает в какой-то директиве идентификатор, он считает его определенным или неопределенным. При этом *определенный* означает, что идентификатор определен препроцессором. Если идентификатор является именем макроса, созданного ранее директивой #define в том же файле, и он не отменялся

посредством #undef, то идентификатор определен. Если идентификатор – не макрос, а, скажем, переменная с областью действия на уровне файла, то с точки зрения препроцессора он не определен.

Определенным может быть объектный макрос, включая пустой макрос, или функциональный макрос:

```
#define LIMIT 1000        // идентификатор LIMIT определен
#define GOOD              // идентификатор GOOD определен
#define A(X) ((-(X))*(X)) // идентификатор A определен
int q;                    // идентификатор q - не макрос, поэтому не определен
#undef GOOD               // идентификатор GOOD не определен
```

Обратите внимание, что область действия макроса #define начинается с места его объявления в файле и продолжается вплоть до соответствующей директивы #undef либо до конца файл в зависимости от того, что случится первым. Кроме того, имейте в виду, что позиция директивы #define в файле будет зависеть от местоположения директивы #include, если макрос поступает из заголовочного файла.

Несколько предопределенных макросов, таких как __DATE__ и __FILE__ (обсуждаются позже в главе), всегда считаются определенными, причем их определение не может быть отменено.

Условная компиляция

Остальные из упомянутых директив можно применять для настройки условной компиляции. Это значит, что их можно использовать для сообщения компилятору о том, принимать либо игнорировать блоки информации или кода согласно условиям на этапе компиляции.

Директивы #ifdef, #else и #endif

Следующий краткий пример прояснит, что делает условная компиляция. Взгляните на следующий код:

```
#ifdef MAVIS
   #include "horse.h"  // выполняется, если идентификатор MAVIS определен
   #define STABLES 5
#else
   #include "cow.h"    // выполняется, если идентификатор MAVIS не определен
   #define STABLES 15
#endif
```

Здесь мы применили отступы, разрешаемые новыми реализациями языка и стандартом ANSI. В случае более старых реализаций может потребоваться выровнять влево все директивы или хотя бы символы #:

```
#ifdef MAVIS
#   include "horse.h"   /* выполняется, если идентификатор MAVIS определен
*/
#   define STABLES   5
#else
#   include "cow.h"    /* выполняется, если идентификатор MAVIS не определен
*/
#   define STABLES  15
#endif
```

Директива #ifdef говорит о том, что если следующий за ней идентификатор (MAVIS) был определен препроцессором, необходимо обработать все директивы и

скомпилировать весь код C до следующей директивы #else или #endif в зависимости от того, что встретится раньше. Если предусмотрена директива #else, то должен быть обработан весь код между #else и #endif, когда идентификатор не определен.

Форма #ifdef #else во многом подобна оператору if else языка C. Основное отличие в том, что препроцессор не распознает фигурные скобки ({ }) как метод обозначения блока, поэтому для пометки блоков директив используются директивы #else (если есть) и #endif (должна присутствовать). Такие условные структуры могут быть вложенными. Как иллюстрируется в листинге 16.9, эти директивы можно применять также для пометки блоков операторов C.

Листинг 16.9. Программа ifdef.c

```
/* ifdef.c -- использование условной компиляции */
#include <stdio.h>
#define JUST_CHECKING
#define LIMIT 4

int main(void)
{
   int i;
   int total = 0;

   for (i = 1; i <= LIMIT; i++)
   {
      total += 2*i*i + 1;
#ifdef JUST_CHECKING
      printf("i=%d, промежуточная сумма = %d\n", i, total);
#endif
   }
   printf("Итоговая сумма = %d\n", total);

   return 0;
}
```

В результате компиляции и выполнения программы будет получен следующий вывод:

```
i=1, промежуточная сумма = 3
i=2, промежуточная сумма = 12
i=3, промежуточная сумма = 31
i=4, промежуточная сумма = 64
Итоговая сумма = 64
```

Если опустить определение JUST_CHECKING (или поместить его в комментарий либо отменить определение с помощью директивы #undef) и повторно скомпилировать программу, отобразится только последняя строка. Таким приемом можно пользоваться, например, при отладке программы.

Определите идентификатор JUST_CHECKING и задействуйте его в условных выборах с помощью #ifdef; компилятор будет включать программный код для вывода промежуточных значений в целях отладки. После отладки определение можно удалить и повторно скомпилировать программу. Если впоследствии снова потребуется вывод промежуточных значений, можно опять вставить определение и избавить себя от необходимости повторно набирать все дополнительные операторы вывода.

Еще одной возможностью является применение #ifdef для выбора альтернативных блоков кода, приспособленных к разным реализациям C.

Директива #ifndef

Директива #ifndef может использоваться совместно с директивами #else и #endif тем же самым способом, что и #ifdef. Директива #ifndef выясняет, *не* определен ли следующий за ней идентификатор; она представляет собой инверсию директивы #ifdef. Эта директива часто применяется для определения константы, если она еще не была определена. Ниже приведен пример.

```
/* arrays.h */
#ifndef SIZE
   #define SIZE 100
#endif
```

(Более старые реализации С могут не разрешать отступ для директивы #define.)

Обычно такая конструкция используется для предотвращения множественных определений одного и того же макроса при включении нескольких заголовочных файлов, каждый из которых может содержать определение. В этом случае определение в первом заголовочном файле становится активным, а последующие определения в других заголовочных файлах игнорируются.

Рассмотрим еще один случай применения. Предположим, что в заголовок файла помещена такая строка:

```
#include "arrays.h"
```

В результате константа SIZE будет определена как 100. Однако если поместить в заголовок файла следующий код:

```
#define SIZE 10
#include "arrays.h"
```

то SIZE устанавливается в 10. Здесь SIZE определяется до обработки файла arrays.h, поэтому строка #define SIZE 100 пропускается. Такой прием можно использовать, например, при тестировании программы с применением массива меньшего размера. Добившись корректной работы программы, можно удалить оператор #define SIZE 10 и провести повторную компиляцию. В этом случае никогда не придется думать о модификации самого заголовочного файла arrays.h.

Директива #ifndef часто используется для предотвращения многократного включения файла. По этой причине заголовочные файлы обычно содержат следующие строки:

```
/* things.h */
#ifndef THINGS_H_
   #define THINGS_H_
   /* остальная часть включаемого файла */
#endif
```

Предположим, что этот файл каким-то образом был включен несколько раз. Когда препроцессор встречает первое включение данного файла, идентификатор THINGS_H_ не определен, поэтому он определяется и обрабатывается остальная часть файла. При появлении следующего включения того же самого файла идентификатор THINGS_H_ уже определен, так что остальная часть файла пропускается.

Из-за чего файл может быть включен несколько раз? Наиболее распространенная причина состоит в том, что многие включаемые файлы содержат директивы включения других файлов, поэтому можно явно включить файл, в котором этот указанный файл уже включен. Почему это является проблемой? Некоторые элементы, помещаемые в заголовочные файлы, такие как объявления типов структур, могут встречать-

ся в файле только один раз. Во избежание многократного включения в стандартных заголовочных файлах применяется директива #ifndef. Одна из задач заключается в том, чтобы удостовериться, что проверяемый идентификатор не определен в другом месте. Поставщики библиотек обычно решают ее путем использования имени файла в качестве идентификатора, записывая имя в верхнем регистре, заменяя точки символами подчеркивания и добавляя символ подчеркивания (или, возможно, два) в качестве префикса и суффикса. Если вы заглянете, скажем, в файл stdio.h, то можете обнаружить в нем примерно такой код:

```
#ifndef _STDIO_H
#define _STDIO_H
// содержимое файла
#endif
```

Вы можете поступать аналогично. Тем не менее, следует избегать применения символа подчеркивания в качестве префикса, т.к. в стандарте указано, что использование подобного года является зарезервированным. Вряд ли вы захотите случайно определить макрос, который конфликтует с чем-либо в стандартных заголовочных файлах. В листинге 16.10 директива #ifndef используется для защиты от многократного включения заголовочного файла из листинга 16.6.

Листинг 16.10. Заголовочный файл names_st.h

```
// names.h -- добавление защиты от многократного включения
#ifndef NAMES_H_
#define NAMES_H_
// константы
#define SLEN 32
// объявления структур
struct names_st
{
   char first[SLEN];
   char last[SLEN];
};
// определения типов
typedef struct names_st names;
// прототипы функций
void get_names(names *);
void show_names(const names *);
char * s_gets(char * st, int n);
#endif
```

Можете протестировать этот заголовочный файл с помощью программы, приведенной в листинге 16.11. Программа должна работать корректно с заголовочным файлом, показанным в листинге 16.10, и не должна успешно компилироваться, если удалить из листинга 16.10 защиту посредством #ifndef.

Листинг 16.11. Программа doubincl.c

```
// doubincl.c -- двукратное включение заголовочного файла
#include <stdio.h>
#include "names.h"
#include "names.h"  // непреднамеренное второе включение
```

```
int main()
{
   names winner = {"Иван", "Иванов"};
   printf("Победителем стал %s %s.\n", winner.first,
         winner.last);
   return 0;
}
```

Директивы #if и #elif

Директива #if во многом похожа на обычный оператор if языка C. За #if следует константное целочисленное выражение, которое считается истинным, когда оно имеет ненулевое значение. В выражении могут применяться логические операции и операции отношения:

```
#if SYS == 1
#include "ibm.h"
#endif
```

Для расширения комбинации #if-#else можно использовать директиву #elif (в некоторых старых реализациях она недоступна). Рассмотрим следующий пример:

```
#if SYS == 1
   #include "ibmpc.h"
#elif SYS == 2
   #include "vax.h"
#elif SYS == 3
   #include "mac.h"
#else
   #include "general.h"
#endif
```

В более новых реализациях предлагается второй способ проверки, определено ли имя. Вместо строки

```
#ifdef VAX
```

можно применять следующую форму записи:

```
#if defined (VAX)
```

Здесь defined — операция препроцессора, которая возвращает значение 1, если ее аргумент определен с помощью директивы #define, и 0 в противном случае. Преимущество такой новой формы состоит в том, что в нее можно использовать вместе с #elif. Исходя из этого, предыдущий пример можно переписать следующим образом:

```
#if defined (IBMPC)
   #include "ibmpc.h"
#elif defined (VAX)
   #include "vax.h"
#elif defined (MAC)
   #include "mac.h"
#else
   #include "general.h"
#endif
```

Если приведенные строки применяются, скажем, в системе VAX, идентификатор VAX должен быть определен где-то раньше в этом файле посредством такой строки:

```
#define VAX
```

Одной из целей использования средств условной компиляции является обеспечение переносимости программы. За счет изменения нескольких ключевых определений в начале файла можно настраивать разные значения и включать определенные файлы для различных систем.

Предопределенные макросы

В стандарте С описано несколько предопределенных макросов, которые перечислены в табл. 16.1.

Таблица 16.1. Предопределенные макросы

Макрос	Описание
`__DATE__`	Строка символов в форме "Ммм дд гггг", представляющая дату обработки препроцессором, например, Aug 24 2014
`__FILE__`	Строка символов, представляющая имя текущего файла исходного кода
`__LINE__`	Целочисленная константа, представляющая номер строки в текущем файле исходного кода
`__STDC__`	Установлен в 1 для указания, что реализация соответствует стандарту С
`__STDC_HOSTED__`	Установлен в 1 для размещаемой среды; в противном случае — 0
`__STDC_VERSION__`	Для С99 установлен в `199901L`; для С11 установлен в `201112L`
`__TIME__`	Время трансляции в форме "чч:мм:сс"

Следует отметить, что стандарт С99 предоставляет предопределенный идентификатор `__func__`, который расширяется до строкового представления имени содержащей его функции. По этой причине данный идентификатор должен иметь область действия в пределах функции, в то время как макросы по существу располагают областью действия на уровне файла. Таким образом, `__func__` является предопределенным идентификатором языка С, а не предопределенным макросом.

В листинге 16.12 демонстрируется ряд предопределенных идентификаторов в действии. Обратите внимание, что некоторые из них являются нововведениями стандарта С99, поэтому компиляторы, разработанные до появления этого стандарта, могут их не принимать. Для компилятора GCC может понадобиться указать флаг `-std=c99` или `-std=c11`.

Листинг 16.12. Программа predef.c

```
// predef.c -- предопределенные идентификаторы
#include <stdio.h>
void why_me();

int main()
{
    printf("Файл: %s.\n", __FILE__);
    printf("Дата: %s.\n", __DATE__);
    printf("Время: %s.\n", __TIME__);
    printf("Версия: is %ld.\n", __STDC_VERSION__);
    printf("Это строка %d.\n", __LINE__);
    printf("Это функция %s\n", __func__);
    why_me();

    return 0;
}
```

```
void why_me()
{
   printf("Это функция %s\n", __func__);
   printf("Это строка %d.\n", __LINE__);
}
```

Вот как выглядит вывод, полученный в результате пробного запуска программы:

```
Файл: predef.c.
Дата: Sep 22 2014.
Время: 22:01:09.
Версия: 201112.
Это строка 11.
Это функция main
Это функция why_me
Это строка 21.
```

Директивы #line и #error

Директива #line позволяет переустанавливать нумерацию строк и имя файла, выводимые с помощью макросов __LINE__ и __FILE__. Директиву #line можно использовать следующим образом:

```
#line 1000       //переустанавливает текущий номер строки в 1000
#line 10 "cool.c" //переустанавливает номер строки в 10, а имя файла – в cool.c
```

Директива #error заставляет препроцессор выдать сообщение об ошибке, которое включает любой текст, указанный в директиве. Если это возможно, процесс компиляции должен приостановиться. Директиву можно применять так:

```
#if __STDC_VERSION__ != 201112L
#error Несоответствие C11
#endif
```

После этого попытка компиляции программы могла бы привести к получению следующих результатов:

```
$ gcc newish.c
newish.c:14:2: error: #error Несоответствие C11
$ gcc -std=c11 newish.c
$
```

Процесс компиляции не проходит, когда компилятор использует более старый стандарт, и завершается успешно, когда применяется стандарт C11.

Директива #pragma

У современных компиляторов существует несколько настроек, которые можно модифицировать с помощью аргументов командной строки или через меню IDE-среды. Директива #pragma позволяет помещать инструкции для компилятора в исходный код. Например, во время разработки стандарта C99 на него ссылались как на C9X, и в одном из компиляторов использовалась следующая директива для включения поддержки этого стандарта:

```
#pragma c9x on
```

В общем случае каждый компилятор имеет собственный набор указаний. Они могут применяться, например, для управления объемом памяти, выделяемой под автоматические переменные, для установки уровня строгости при проверке ошибок или

для включения нестандартных языковых средств. В стандарте C99 предоставляются три стандартных указания технической природы, которые здесь не рассматриваются.

Кроме того, стандарт C99 поддерживает операцию препроцессора _Pragma. Она преобразует строку в обычное указание компилятору.

Например, операция

```
_Pragma("nonstandardtreatmenttypeB on")
```

является эквивалентом следующего указания:

```
#pragma nonstandardtreatmenttypeB on
```

Поскольку в этой операции не используется символ #, она может выступать в качестве части расширения макроса:

```
#define PRAGMA(X) _Pragma(#X)
#define LIMRG(X) PRAGMA(STDC CX_LIMITED_RANGE  X)
```

После этого можно применять код вроде показанного ниже:

```
LIMRG ( ON )
```

Кстати, следующее определение не работает, хотя выглядит вполне корректным:

```
#define LIMRG(X) _Pragma(STDC CX_LIMITED_RANGE  #X)
```

Проблема в том, что оно полагается на конкатенацию строк, но компилятор не выполняет конкатенацию до тех пор, пока не завершится работа препроцессора.

Оператор _Pragma выполняет всю работу по превращению из строк, т.е. управляющие последовательности в строке преобразуются в представляющие их символы. Таким образом, вызов операции

```
_Pragma("use_bool \"true \"false")
```

принимает следующий вид:

```
#pragma use_bool "true "false
```

Обобщенный выбор (C11)

Термин *обобщенное программирование* относится к коду, который не является специфичным для конкретного типа, но после указания типа может транслироваться в код для этого типа. Например, язык C++ позволяет создавать обобщенные алгоритмы в форме шаблонов, которые компилятор затем использует при автоматическом создании экземпляра кода для указанного типа. В языке C нет ничего близко похожего на это. Тем не менее, в C11 появился новый вид выражения, называемого *выражением обобщенного выбора*, которое можно применять для выбора значения на основе типа выражения, т.е. базируясь на том, является ли типом выражения int, double и т.д. Выражение обобщенного выбора — это не оператор препроцессора, но обычно оно используется как часть определения макроса #define, обладающего определенными чертами обобщенного программирования.

Выражение обобщенного выбора выглядит следующим образом:

```
_Generic(x, int: 0, float: 1, double: 2, default: 3)
```

Здесь _Generic — новое ключевое слово C11. Круглые скобки после _Generic содержат несколько элементов, разделенных запятыми. Первый элемент представляет собой выражение, а каждый из оставшихся элементов — тип, за которым следует значение, наподобие float: 1. Тип первого элемента соответствует одной из меток, и значением всего выражения будет значение, указанное после давшей совпадение метки.

Например, предположим, что x в показанном выше выражении является переменной типа int. Тогда тип x соответствует метке int:, приводя к тому, что все выражение получает значение 0. Если тип не соответствует ни одной метке, значением всего выражения становится то, что указано после метки default:. Оператор обобщенного выбора немного похож на оператор switch за исключением того, что сопоставление с метками производится для типа выражения, а не его значения.

Давайте рассмотрим пример объединения оператора обобщенного выбора с определение макроса:

```
#define MYTYPE(X) _Generic((X),\
    int: "int",\
    float : "float",\
    double: "double",\
    default: "other"\
)
```

Вспомните, что макрос должен быть определен в одной логической строке, но с помощью символа \ одну логическую строку можно разбивать на несколько физических строк. В данном случае выражение обобщенного выбора оценивается как строка. Скажем, вызов макроса MYTYPE(5) оценивается как строка "int", поскольку тип значения 5 соответствует метке int:. В листинге 16.13 приведена дальнейшая иллюстрация этого макроса.

Листинг 16.13. Программа predef.c

```
// mytype.c
#include <stdio.h>
#define MYTYPE(X) _Generic((X),\
   int: "int",\
   float : "float",\
   double: "double",\
   default: "другой"\
)
int main(void)
{
   int d = 5;
   printf("%s\n", MYTYPE(d));     // d имеет тип int
   printf("%s\n", MYTYPE(2.0*d)); // 2.0* d имеет тип double
   printf("%s\n", MYTYPE(3L));    // 3L имеет тип long
   printf("%s\n", MYTYPE(&d));    // &d имеет тип int *
   return 0;
}
```

Вот вывод программы:

```
int
double
другой
другой
```

В последних двух обращениях к MYTYPE() используются типы, не имеющие соответствующих меток, поэтому выбирается строка с меткой default:. Мы могли бы предусмотреть большее число меток, расширив возможности макроса, но этот пример задуман только в качестве демонстрации особенностей работы макросов, основанных на _Generic.

При оценке выражения обобщенного выбора программа не вычисляет первый элемент; она только выясняет его тип. Единственным вычисляемым выражением является то, которое указано в совпадающей метке.

Средство _Generic можно применять для определения макросов, которые действуют подобно функциям, не зависящим от типа ("обобщенным"). В разделе, посвященном библиотеке math, далее в главе будет приведен пример.

Встраиваемые функции (C99)

Обычно с вызовом функции связаны накладные расходы. Это означает, что подготовка вызова, передача аргументов, переход к коду функции и возврат требуют времени на выполнение. Как вы уже видели, макрос можно использовать для встраивания кода, тем самым избегая таких накладных расходов. В стандарте C99 был позаимствован у C++ (но не во всем точно) другой подход — *встраиваемые функции*. Исходя из его названия, вы могли бы ожидать, что встраиваемая функция заменяет вызов функции встраиваемым кодом, но это не так. В стандартах C99 и C11 на самом деле указано так: "превращение функции во встраиваемую предполагает, что ее вызов будет настолько быстрым, насколько это возможно. Степень, до которой подобные предположения эффективны, зависит от реализации". Таким образом, преобразование функции во встроенную может привести к тому, что компилятор заменит вызов функции встраиваемым кодом и/или предпримет оптимизации другого рода либо вообще не окажет никакого воздействия.

Существуют разные способы создания определений встраиваемых функций. В стандарте говорится о том, что функция с внутренним связыванием может быть сделана встраиваемой, и данное определение для встраиваемой функции должно находиться в том же файле, где функция применяется. Поэтому простой подход предполагает использование спецификатора функции inline наряду со спецификатором класса хранения static. Как правило, встраиваемые функции определяются до их первого применения в файле, так что определение действует также и в качестве прототипа. Другими словами, код будет выглядеть примерно так:

```
#include <stdio.h>
inline static void eatline()  // встраиваемое определение/прототип
{
   while (getchar() != '\n')
       continue;
}
int main()
{
   ...
   eatline();                 // вызов функции
   ...
}
```

Встретив встраиваемое объявление, компилятор может, к примеру, заменить вызов функции eatline() ее телом. Это значит, что результат может быть такой, как если бы вы взамен написали следующий код:

```
#include <stdio.h>
inline static void eatline()  // встраиваемое определение/прототип
{
   while (getchar() != '\n')
       continue;
}
```

```
int main()
{
   ...
   while (getchar() != '\n')   // замена вызова функции
       continue;

}
```

Поскольку встраиваемая функция не имеет отдельного предназначенного для нее блока кода, получить ее адрес нельзя. (В действительности это возможно, но тогда компилятор сгенерирует функцию, отличную от встраиваемой.) Кроме того, встраиваемая функция может быть не видна в отладчике.

Встраиваемая функция должна быть короткой. Для длинной функции время, затрачиваемое на ее вызов, невелико по сравнению со временем выполнения тела функции, поэтому использование встраиваемой функции не обеспечит существенной экономии времени.

Для проведения оптимизаций по встраиванию функции компилятору должно быть известно содержимое определения функции. Это означает, что определение встраиваемой функции должно находиться в том же файле, что и ее вызов. По данной причине встраиваемая функция обычно имеет внутреннее связывание. Следовательно, если программа состоит из нескольких файлов, встраиваемое определение понадобится поместить в каждый файл, который вызывает функцию. Для достижения такого условия проще всего указать определение встраиваемой функции в заголовочном файле и затем включать этот файл в файлы, где функция применяется.

```
// eatline.h
#ifndef EATLINE_H_
#define EATLINE_H_
inline static void eatline()
{
   while (getchar() != '\n')
       continue;
}
#endif
```

Встраиваемая функция является исключением из правила, которое не рекомендует помещать исполняемый код в заголовочный файл. Так как встраиваемая функция имеет внутреннее связывание, ее определение в нескольких файлах не вызывает проблем.

В отличие от C++, язык C разрешает также смешивать встраиваемые определения с внешними определениями (определениями функций с внешним связыванием). Например, рассмотрим программу, состоящую из следующих трех файлов:

```
// file1.c
...
inline static double square(double);
double square(double x) { return x * x; }
int main()
{
   double q = square(1.3);
   ...
// file2.c
...
double square(double x) { return (int) (x*x); }
```

```
void spam(double v)
{
    double kv = square(v);
    ...
// file3.c
...
inline double square(double x) { return (int) (x * x + 0.5); }
void masp(double w)
{
    double kw = square(w);
    ...
```

Первый файл содержит определение inline static, как и ранее. Второй файл имеет определение обычной функции, отсюда и наличие внешнего связывания. Третий файл включает определение inline, в котором не указан квалификатор static.

Что здесь происходит? Функция spam() в file2.c использует определение square() из этого файла. Данное определение, имея внешнее связывание, является видимым другим файлам, но main() в file1.c применяет локальное определение static функции square(). Поскольку это определение также inline, компилятор может (или нет) оптимизировать код, возможно, встроив его. Наконец, для file3.c компилятор свободен в использовании либо встраиваемого определения из file3.c, либо определения с внешним связыванием из file2.c (или обоих!). Если вы не укажете static в определении inline, как в файле file3.c, то определение inline рассматривается в качестве альтернативы, которая могла бы применяться вместо внешнего определения.

Обратите внимание, что до появления C99 встраиваемые функции в GCC были реализованы с использованием несколько отличающихся правил, так что интерпретация GCC спецификатора inline может зависеть от указанных флагов компилятора.

Функции _Noreturn (C11)

Когда в стандарте C99 появилось ключевое слово inline, оно было единственным примером спецификатора функции. (Ключевые слова extern и static называются спецификаторами класса хранения и могут применяться к объектам данных, а также к функциям.) В стандарт C11 был добавлен второй спецификатор функции, _Noreturn, предназначенный для указания функции, которая по завершении не возвращает управление вызывающей функции. Примером функции _Noreturn является exit(); после обращения к ней вызывающая функция никогда не возобновит свое выполнение. Обратите внимание, что это отличается от возвращаемого типа void. Типичная функция void возвращает управление вызывающей функции; она просто не предоставляет какое-либо значение.

Цель _Noreturn заключается в том, чтобы проинформировать пользователя и компилятор, что конкретная функция не возвратит управление вызывающей программе. Информирование пользователя помогает предотвратить неправильное употребление функции, а указание на такой факт компилятору может сделать возможными некоторые оптимизации кода.

Библиотека C

Первоначально официальной библиотеки C не существовало. Позже возник стандарт де-факто, основанный на реализации C для Unix. Комитет ANSI C, в свою оче-

редь, разработал официальную стандартную библиотеку, которая в значительной степени базировалась на этом стандарте де-факто. Учитывая распространение языка С по всему миру, комитет затем решил переопределить библиотеку, чтобы она могла быть реализована в широком разнообразии систем.

Мы уже обсуждали некоторые функции ввода-вывода, функции для обработки символов и функции для работы со строками из этой библиотеки. В данной главе мы исследуем еще несколько функций, но сначала поговорим о том, как использовать библиотеку.

Получение доступа к библиотеке С

Способ получения доступа к библиотеке С зависит от реализации языка, поэтому вам необходимо ознакомиться с тем, насколько более общие утверждения применимы к вашей системе. Во-первых, библиотечные функции часто можно обнаружить в ряде разных мест. Например, функция `getchar()` обычно определена в виде макроса внутри `stdio.h`, но функция `strlen()`, как правило, содержится в библиотечном файле. Во-вторых, для разных систем предусмотрены отличающиеся способы получения доступа к этим функциям. В последующих разделах в общих чертах представлены три возможности.

Автоматический доступ

Во многих системах достаточно всего лишь скомпилировать программу, т.к. многие распространенные библиотечные функции сделаны доступными автоматически.

Имейте в виду, что для используемых функций вы должны объявить их типы. Обычно это можно сделать путем включения подходящего заголовочного файла. Файлы, подлежащие включению, описаны в руководствах пользователя по библиотечным функциям. Однако в некоторых старых системах могла возникать необходимость в самостоятельном наборе объявлений функций. В этом случае тип функции снова следует искать в руководстве пользователя. Кроме того, в приложении Б приведено описание библиотеки ANSI С, группирующее функции по заголовочным файлам.

В прошлом имена заголовочных файлов не были согласованы между разными реализациями. Стандарт ANSI С группирует библиотечные функции в семейства. Для каждого семейства предусмотрен заголовочный файл с прототипами функций.

Включение файлов

Если функция определена в виде макроса, то с помощью директивы `#include` можно включить файл, содержащий ее определение. Часто похожие макросы собираются в заголовочный файл с подходящим именем. Например, с появлением стандарта ANSI С компиляторы С поступают с файлом `ctype.h`, содержащим ряд макросов, которые определяют природу символа: верхний регистр, цифра и т.п.

Включение библиотек

На определенном этапе компиляции либо компоновки программы может понадобиться указать опцию библиотеки. Даже система, которая автоматически проверяет свою стандартную библиотеку, может иметь другие библиотеки функций, используемые менее часто. Эти библиотеки должны запрашиваться явно с применением опций компилятора. Обратите внимание, что данный процесс отличается от включения заголовочного файла. Заголовочный файл предоставляет объявления или прототипы функций. Опция библиотеки сообщает системе, где искать код функций. Очевидно, мы не можем пройтись по особенностям всех систем, но настоящее обсуждение поможет понять, на что обращать внимание.

Использование описаний библиотеки

Ограниченный объем книги не позволяет обсудить библиотеки полностью, но мы рассмотрим некоторые характерные примеры. Для начала обратимся к документации.

Документацию по функциям можно найти в нескольких местах. Система может иметь онлайновое руководство, а IDE-среда часто располагает онлайновой справкой. Поставщики компиляторов С иногда предоставляют руководства пользователя в печатном виде, которые содержат описание библиотечных функций, либо компакт-диск с аналогичным материалом. Многие издательства выпустили справочные пособия по функциям библиотеки С. Одни из них имеют общую природу, а другие ориентированы на определенные реализации языка. Кроме того, как упоминалось ранее, краткое описание функций содержится в приложении Б настоящей книги.

При чтении документации важно понимать заголовки функций. Описание со временем меняется. Для примера рассмотрим описание функции `fread()` в старой документации для Unix:

```
#include <stdio.h>

fread(ptr, sizeof(*ptr), nitems, stream)
FILE *stream;
```

Сначала включается подходящий файл. Типы для `fread()`, `ptr`, `sizeof(*ptr)` и `nitems` не указаны. В то время для этих элементов по умолчанию принимался тип `int`, но контекст проясняет тот факт, что `ptr` является указателем. (В ранних версиях С указатели обрабатывались как целые числа.) Аргумент `stream` объявлен как указатель на `FILE`. Объявление создает впечатление, что в качестве второго аргумента применяется операция `sizeof`. В действительности здесь указано, что значением этого аргумента должен быть размер объекта, указанного с помощью `ptr`. Часто будет использоваться операция `sizeof`, как показано выше, но с точки зрения синтаксиса допускается любое значение типа `int`.

Позже форма изменилась следующим образом:

```
#include <stdio.h>

int fread(ptr, size, nitems, stream;)
char *ptr;
int size, nitems;
FILE *stream;
```

Теперь все типы данных заданы явно, а `ptr` трактуется как указатель на `char`.

Стандарт ANSI C90 предоставляет такое описание:

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Во-первых, в нем применяется новый формат прототипа. Во-вторых, некоторые типы изменились. Тип `size_t` определен как целочисленный тип без знака, возвращаемый операцией `sizeof`. Обычно им будет либо `unsigned int`, либо `unsigned long`. Файл `stddef.h` содержит определение `typedef` или `#define` для `size_t`, как и несколько других файлов, в том числе `stdio.h`, обычно за счет включения `stddef.h`. Многие функции, включая `fread()`, часто встраивают операцию `sizeof` в виде части фактического аргумента. Тип `size_t` обеспечивает соответствие формального аргумента этому общему способу использования.

Кроме того, в ANSI С применяется указатель на `void` в качестве своего рода обобщенного указателя для ситуаций, когда могут использоваться указатели на разные типы данных.

Например, первым аргументом функции `fread()` может быть указатель на массив значений `double` или на некоторую структуру. Если фактический аргумент представляет собой указатель, скажем, на массив из 20 значений `double`, а формальный аргумент является указателем на `void`, компилятор примет вариант для подходящего типа и не уведомит о несоответствии типов.

Относительно недавно стандарты C99/C11 внедрили в описание функций новое ключевое слово `restrict`:

```
#include <stdio.h>
size_t fread(void * restrict ptr, size_t size,
             size_t nmemb, FILE * restrict stream);
```

А теперь давайте перейдем к обзору некоторых специфических функций.

Библиотека математических функций

Библиотека математических функций содержит множество удобных функций такого рода. Их объявления или прототипы содержатся в заголовочном файле `math.h`. В табл. 16.2 перечислено несколько функций, объявленных в `math.h`. Обратите внимание, что все углы измеряются в радианах (один радиан составляет $180/\pi = 57.296$ градуса). В разделе V приложения Б представлен полный список функций, определенных стандартом C99.

Таблица 16.2. Некоторые стандартные математические функции ANSI C

Прототип	Описание
`double acos(double x)`	Возвращает угол (от 0 до π радиан), косинус которого равен x
`double asin(double x)`	Возвращает угол (от $-\pi/2$ до $\pi/2$ радиан), синус которого равен x
`double atan(double x)`	Возвращает угол (от $-\pi/2$ до $\pi/2$ радиан), тангенс которого равен x
`double atan2(double y, double x)`	Возвращает угол (от $-\pi$ до π радиан), тангенс которого равен y/x
`double cos(double x)`	Возвращает косинус x (x в радианах)
`double sin(double x)`	Возвращает синус x (x в радианах)
`double tan(double x)`	Возвращает тангенс x (x в радианах)
`double exp(double x)`	Возвращает экспоненциальную функцию x (e^x)
`double log(double x)`	Возвращает натуральный логарифм x
`double log10(double x)`	Возвращает логарифм x по основанию 10
`double pow(double x, double y)`	Возвращает x в степени y
`double sqrt(double x)`	Возвращает квадратный корень x
`double cbrt(double x)`	Возвращает кубический корень x
`double ceil(double x)`	Возвращает наименьшее целое, которое не меньше x
`double fabs(double x)`	Возвращает абсолютное значение x
`double floor(double x)`	Возвращает наибольшее целое, которое не больше x

Немного тригонометрии

Воспользуемся библиотекой математических функций для решения типичной задачи преобразования прямоугольных координат в полярные (модуль и угол). Предположим, что на сетке проведена линия, протяженность которой составляет 4 единицы по горизонтали (значение x) и 3 единицы по вертикали (значение y). Каковы длина (модуль) и направление линии? Согласно тригонометрии:

$$\textit{модуль} = \text{квадратный корень } (x^2 + y^2)$$

и

$$\textit{угол} = \text{арктангенс } (y/x)$$

Библиотека math предоставляет функцию извлечения квадратного корня и пару функций вычисления арктангенса, поэтому вы можете выразить данное решение на языке С. Функция извлечения квадратного корня, sqrt(), принимает аргумент double и возвращает квадратный корень аргумента также в виде значения double.

Функция atan() принимает аргумент double и возвращает угол, значение тангенса которого равно этому аргументу. К сожалению, функция atan() не учитывает квадрант вектора. Например, если координаты x и y вектора равны -5 и -5, функция atan() даст результат 45°, поскольку (-5)/(-5) = 1. Тот же результат будет для вектора с координатами 5 и 5. Другими словами, функция atan() не различает векторы, углы которых отличаются на 180°. (На самом деле функция atan() выводит результат в радианах, а не в градусах; мы обсудим это преобразование позже.)

К счастью, библиотека С содержит и функцию atan2(). Она принимает два аргумента: значения x и y. Таким образом, функция способна анализировать знаки координат и правильно определять угол. Подобно atan(), функция atan2() возвращает угол в радианах.

Чтобы преобразовать радианы в градусы, умножьте результирующий угол на 180 и разделите на π. Вычисление значения ϖ можно поручить компьютеру, указав выражение 4 * atan(1). Все описанные действия продемонстрированы в листинге 16.14. Вдобавок у вас есть шанс освежить в памяти знания структур и typedef.

Листинг 16.14. Программа rect_pol.c

```
/* rect_pol.c -- преобразует прямоугольные координаты в полярные */
#include <stdio.h>
#include <math.h>

#define RAD_TO_DEG (180/(4 * atan(1)))

typedef struct polar_v {
   double magnitude;
   double angle;
} Polar_V;

typedef struct rect_v {
   double x;
   double y;
} Rect_V;

Polar_V rect_to_polar(Rect_V);

int main(void)
{
   Rect_V input;
   Polar_V result;

   puts("Введите координаты x и y; введите q для выхода:");
```

```
    while (scanf("%lf %lf", &input.x, &input.y) == 2)
    {
        result = rect_to_polar(input);
        printf("модуль = %0.2f, угол = %0.2f\n",
               result.magnitude, result.angle);
    }
    puts("Программа завершена.");

    return 0;
}

Polar_V rect_to_polar(Rect_V rv)
{
    Polar_V pv;

    pv.magnitude = sqrt(rv.x * rv.x + rv.y * rv.y);
    if (pv.magnitude == 0)
        pv.angle = 0.0;
    else
        pv.angle = RAD_TO_DEG * atan2(rv.y, rv.x);

    return pv;
}
```

Ниже показаны результаты пробного запуска:

```
Введите координаты x и y; введите q для выхода:
10 10
модуль = 14.14, угол = 45.00
-12 -5
модуль = 13.00, угол = -157.38
q
Программа завершена.
```

Если в процессе компиляции будет выдано сообщение, такое как

```
Undefined: _sqrt
```
Не определено: _sqrt

или

```
'sqrt': unresolved external
```
'sqrt': нераспознанный внешний идентификатор

либо нечто подобное, значит, компилятор-компоновщик не смог найти библиотеку математических функций. В системах Unix может потребоваться указать компоновщику на необходимость поиска библиотеки математических функций с помощью флага `-lm`:

```
cc rect_pol.c -lm
```

Обратите внимание, что флаг `-lm` находится в конце команды. Причина в том, что компоновщик вступает в игру после того, как компилятор скомпилирует файл С. Компилятор GCC в системе Linux может вести себя в такой же манере:

```
gcc rect_pol.c -lm
```

Варианты типов

Базовые математические функции с плавающей запятой принимают аргументы типа `double` и возвращают значение типа `double`. Им можно передавать аргументы типа `float` или `long double`, и функции по-прежнему будут работать, поскольку аргументы указанных типов преобразуются в тип `double`. Это удобно, но не обязательно

оптимально. Если двойная точность не нужна, то вычисления могут выполняться быстрее, если применять значения float с одинарной точностью. К тому же значение типа long double будет терять точность при передаче параметру типа double; может даже оказаться, что значение вообще непредставимо. Чтобы решить такие потенциальные проблемы, в стандарте C предоставляются версии стандартных функций типа float и типа long double, имеющие в имени суффикс f или l (строчная буква "L"). Таким образом, sqrtf() — это версия типа float функции sqrt(), a sqrtl() — версия типа long double функции sqrt().

Появление в C11 выражения обобщенного выбора позволяет определять обобщенный макрос, который выбирает наиболее подходящую версию математической функции на основе типа аргумента. В листинге 16.15 продемонстрированы два подхода.

Листинг 16.15. Программа generic.c

```
// generic.c -- определение обобщенных макросов
#include <stdio.h>
#include <math.h>
#define RAD_TO_DEG (180/(4 * atanl(1)))
// обобщенная функция извлечения квадратного корня
#define SQRT(X) _Generic((X),\
   long double: sqrtl,\
   default: sqrt,\
   float: sqrtf)(X)
// обобщенная функция вычисления синуса угла, заданного в градусах
#define SIN(X) _Generic((X),\
   long double: sinl((X)/RAD_TO_DEG),\
   default:     sin((X)/RAD_TO_DEG),\
   float:       sinf((X)/RAD_TO_DEG)\
)
int main(void)
{
   float x = 45.0f;
   double xx = 45.0;
   long double xxx =45.0L;
   long double y = SQRT(x);
   long double yy= SQRT(xx);
   long double yyy = SQRT(xxx);
   printf("%.17Lf\n", y);        // соответствует float
   printf("%.17Lf\n", yy);       // соответствует default
   printf("%.17Lf\n", yyy);      // соответствует long double
   int i = 45;
   yy = SQRT(i);                 // соответствует default
   printf("%.17Lf\n", yy);
   yyy= SIN(xxx);                // соответствует long double
   printf("%.17Lf\n", yyy);
   return 0;
}
```

Вывод выглядит следующим образом:

```
6.70820379257202148
6.70820393249936942
6.70820393249936909
6.70820393249936942
0.70710678118654752
```

Как видите, SQRT(i) имеет такое же возвращаемое значение, как у SQRT(xx), поскольку типы обоих аргументов (int и double) соответствуют метке default.

Интересно взглянуть на то, каким образом заставить макрос, использующий _Generic, действовать подобно функции. В определении SIN() предпринят, вероятно, наиболее очевидный подход: каждое помеченное значение представляет собой вызов функции, поэтому значением выражения _Generic является отдельный вызов функции, такой как sinf((X)/RAD_TO_DEG), с аргументом SIN(), заменяющим X.

Определение SQRT(), пожалуй, более элегантно. В этом случае значение выражения _Generic — это имя функции, такое как sinf. Это имя функции заменяется ее адресом, так что значением выражения _Generic будет указатель на функцию. Однако за полным выражением _Generic следует (X), и комбинация *указатель-на-функцию(аргумент)* вызывает указанную функцию с заданным аргументом.

Говоря кратко, для SIN() вызов функции находится внутри выражения обобщенного выбора, в то время как для SQRT() выражение обобщенного выбора оценивается как указатель, который затем применяется для вызова функции.

Библиотека tgmath.h (C99)

Стандарт C99 предлагает заголовочный файл tgmath.h, в котором определены макросы обобщенного типа, по своему действию похожие на те, что были показаны в листинге 16.15. Если какая-то функция math.h определена для каждого из трех типов float, double и long double, то файл tgmath.h создает макрос обобщенного типа с тем же именем, что и у версии для double. Например, он определяет макрос sqrt(), который разворачивается в функцию sqrtf(), sqrt() или sqrtl() в зависимости от типа предоставленного аргумента. Другими словами, макрос sqrt() ведет себя подобно макросу SQRT() из листинга 16.15.

Если компилятор поддерживает арифметику комплексных чисел, то в нем доступен файл complex.h, в котором объявлены комплексные аналоги математических функций. Например, в этом файле объявлены функции csqrtf(), csqrt() и csqrtl(), которые возвращают комплексный квадратный корень типа float complex, double complex и long double complex, соответственно. Когда такая поддержка предоставляется, макрос sqrt() из tgmath.h также может разворачиваться в связанную функцию комплексного квадратного корня.

Если вы хотите, скажем, вызвать функцию sqrt() вместо макроса sqrt(), даже несмотря на то, что файл tgmath.h включен, можете поместить имя функции в круглые скобки:

```
#include <tgmath.h>
   ...
   float x = 44.0;
   double y;
   y = sqrt(x);       // вызов макроса, следовательно sqrtf(x)
   y = (sqrt)(x);     // вызов функции sqrt()
```

Код работает, поскольку имя функционального макроса должно сопровождаться последующей открывающей круглой скобкой, что обходится путем заключения имени в пару скобок. В противном случае круглые скобки не оказывают воздействия на выражение, находящееся внутри них, кроме изменения порядка следования операций, поэтому помещение в скобки имени функции по-прежнему в результате приводит к ее вызову. На самом деле, из-за странно противоречивых правил, принятых в C относительно указателей на функции, для вызова функции sqrt() можно также использовать (*sqrt)().

Средство выражений _Generic, добавленное в стандарт C11, является простым способом реализации макросов tgmath.h, не прибегая к механизмам, которые выходят за рамки стандарта C.

Библиотека утилит общего назначения

Библиотека утилит общего назначения содержит множество функций, включая генератор случайных чисел, функции для поиска и сортировки, функции для преобразования и функции для управления памятью. Вы уже видели работу функций rand(), srand(), malloc() и free() в главе 12. В ANSI C прототипы этих функций находятся в заголовочном файле stdlib.h. В разделе V приложения Б перечислены все функции в этом семействе; далее мы рассмотрим некоторые из них более подробно.

Функции exit() и atexit()

Функция exit() уже применялась явно в нескольких примерах. Вдобавок функция exit() вызывается автоматически при возвращении из main(). В стандарт ANSI добавлена пара интересных возможностей, которые мы еще не использовали. Самая важная из них – возможность указания определенных функций, которые должны вызываться во время выполнения exit(). Функция atexit() обеспечивает это за счет регистрации функций, предназначенных для вызова при выполнении exit(). Функция atexit() принимает в качестве аргумента указатель на функцию.

В листинге 16.16 показано, как это работает.

Листинг 16.16. Программа byebye.c

```
/* byebye.c -- пример применения atexit() */
#include <stdio.h>
#include <stdlib.h>
void sign_off(void);
void too_bad(void);

int main(void)
{
    int n;

    atexit(sign_off);    /* регистрация функции sign_off() */
    puts("Введите целое число:");
    if (scanf("%d",&n) != 1)
    {
        puts("Это не целое число!");
        atexit(too_bad);  /* регистрация функции too_bad() */
        exit(EXIT_FAILURE);
    }
    printf("%d является %s.\n", n,  (n % 2 == 0)? "четным" : "нечетным");

    return 0;
}
void sign_off(void)
{
    puts("Завершение работы очередной замечательной программы от");
    puts("SeeSaw Software!");
}
void too_bad(void)
{
    puts("SeeSaw Software приносит искренние соболезнования");
    puts("в связи с отказом программы.");
}
```

Ниже показаны результаты пробного запуска:

```
Введите целое число:
212
212 является четным.
Завершение работы очередной замечательной программы от
SeeSaw Software!
```

Если программа выполняется в IDE-среде, две последних строки вы можете не увидеть.

Рассмотрим результаты второго пробного запуска:

```
Введите целое число:
что?
Это не целое число!
SeeSaw Software приносит искренние соболезнования
в связи с отказом программы.
Завершение работы очередной замечательной программы от
SeeSaw Software!
```

Если программа выполняется в IDE-среде, четыре последних строки вы можете не увидеть.

Давайте посмотрим на две основных области: применение функции `atexit()` и аргументы `exit()`.

Использование функции `atexit()`

Все-таки есть функция, которая принимает указатели на функции! Чтобы использовать ее, просто передайте адрес функции, которая должна быть вызвана при выходе. Поскольку имя функции действует как адрес, когда применяется в качестве аргумента функции, для аргумента можно указывать имя `sign_off` или `too_bad`. Затем `atexit()` регистрирует эту функцию в списке функций, предназначенных для выполнения при вызове `exit()`. Стандарт ANSI гарантирует, что список может вмещать не менее 32 функций. Каждая из них добавляется с помощью отдельного вызова `atexit()`. Когда функция `exit()`, в конце концов, вызывается, она выполняет эти функции, причем первой выполняется функция, добавленная в список последней.

Обратите внимание, что в результате недопустимого ввода пользователя вызываются обе функции, `sign_off()` и `too_bad()`, но в случае ввода допустимого значения вызывается только `sign_off()`. Дело в том, что оператор `if` регистрирует функцию `too_bad()` только для случая недопустимого ввода. Кроме того, первой была вызвана функция, зарегистрированная последней.

Функции, регистрируемые `atexit()`, вроде `sign_off()` и `too_bad()`, должны иметь тип `void` и не принимать аргументов. Обычно они решают вспомогательные задачи, такие как обновление файла мониторинга программы или переустановка переменных среды.

Обратите внимание, что `sign_off()` вызывается даже в случае, когда функция `exit()` не вызывается явно; причина в том, что `exit()` неявно вызывается при завершении `main()`.

Использование функции `exit()`

Когда `exit()` выполняет функции, указанные с помощью `atexit()`, она предпринимает собственные шаги по очистке. Функция `exit()` сбрасывает все потоки вывода, закрывает все открытые потоки и закрывает временные файлы, созданные в результате обращений к стандартной функции ввода-вывода `tmpfile()`. Затем `exit()` возвращает управление размещаемой среде и по возможности сообщает среде состояние

завершения. Традиционно программы для Unix применяли 0, чтобы указать на успешное завершение, и ненулевое значение для сообщения об отказе. Коды возврата Unix не обязательно работают со всеми системами, поэтому в ANSI C определен макрос по имени EXIT_FAILURE, который может использоваться переносимым образом для обозначения отказа. Аналогично, в ANSI C определен макрос EXIT_SUCCESS для указания на успешное завершение, но exit() также принимает для этой цели значение 0. В рамках стандарта ANSI C применение exit() в нерекурсивной функции main() эквивалентно использованию ключевого слова return. Тем не менее, exit() также завершает программу, когда применяется в функциях, отличных от main().

Функция qsort()

Метод быстрой сортировки входит в число наиболее эффективных алгоритмов сортировки, особенно в случае крупных массивов. Разработанный Чарльзом Энтони Ричардом Хоаром в 1962 году, этот алгоритм разделяет массивы на постоянно уменьшающиеся части, пока не будет достигнут уровень элемента. Сначала массив делится на две части, так что любое значение в одной части меньше любого значения в другой части. Этот процесс продолжается вплоть до момента, когда массив станет полностью отсортированным.

Алгоритм быстрой сортировки реализован в C под именем qsort(). Эта функция сортирует массив объектов данных. Она имеет следующий прототип ANSI:

```
void qsort (void *base, size_t nmemb, size_t size,
        int (*compar)(const void *, const void *));
```

Первый аргумент представляет собой указатель на начало сортируемого массива. Стандарт ANSI C разрешает приведение типа указателя на данные к типу указателя на void, таким образом, позволяя первому фактическому аргументу qsort() ссылаться на массив любого вида.

Во втором аргументе передается количество элементов, подлежащих сортировке. Прототип преобразует это значение в тип size_t. Как уже несколько раз упоминалось, size_t является целочисленным типом, который возвращается операцией sizeof и определен в стандартных заголовочных файлах.

Из-за того, что функция qsort() преобразует свой первый аргумент в указатель на void, она утрачивает информацию о размере каждого элемента массива. Чтобы скомпенсировать это, следует явно сообщить qsort() размер объекта данных. Именно для такой цели служит третий аргумент. Например, если вы сортируете массив типа double, то в третьем аргументе должны указать sizeof(double).

Наконец, qsort() требуется указатель на функцию, которая будет использоваться для определения порядка сортировки. Функция сравнения должна принимать два аргумента – указатели на два сравниваемых элемента. Она возвращает положительное целое число, если первый элемент должен следовать за вторым, ноль, если элементы одинаковы, и отрицательное целое число, если второй элемент должен следовать за первым. Функция qsort() вызывает эту функцию с передачей ей значений указателей, которые вычисляет на основе другой предоставленной информации.

Форма функции сравнения задана в последнем аргументе прототипа qsort():

```
int (*compar)(const void *, const void *)
```

Здесь видно, что последний аргумент qsort() представляет собой указатель на функцию, возвращающую значение int и принимающую два аргумента, каждый из которых является указателем на тип const void. Эти два указателя ссылаются на сравниваемые элементы.

Листинг 16.17 и последующее обсуждение иллюстрируют способ определения функции сравнения и применения функции qsort(). В программе создается массив случайных значений с плавающей запятой, который затем сортируется.

Листинг 16.17. Программа qsorter.c

```
/* qsorter.c -- использование быстрой сортировки для упорядочения групп чисел */
#include <stdio.h>
#include <stdlib.h>

#define NUM 40
void fillarray(double ar[], int n);
void showarray(const double ar[], int n);
int mycomp(const void * p1, const void * p2);

int main(void)
{
    double vals[NUM];
    fillarray(vals, NUM);
    puts("Список случайных чисел:");
    showarray(vals, NUM);
    qsort(vals, NUM, sizeof(double), mycomp);
    puts("\nОтсортированный список:");
    showarray(vals, NUM);
    return 0;
}

void fillarray(double ar[], int n)
{
    int index;

    for( index = 0; index < n; index++)
        ar[index] = (double)rand()/((double) rand() + 0.1);
}

void showarray(const double ar[], int n)
{
    int index;

    for( index = 0; index < n; index++)
    {
        printf("%9.4f ", ar[index]);
        if (index % 6 == 5)
            putchar('\n');
    }
    if (index % 6 != 0)
        putchar('\n');
}

/* сортировка по возрастанию */
int mycomp(const void * p1, const void * p2)
{
    /* для доступа к значениям необходимо использовать указатели на double */
    const double * a1 = (const double *) p1;
    const double * a2 = (const double *) p2;

    if (*a1 < *a2)
        return -1;
    else if (*a1 == *a2)
        return 0;
    else
        return 1;
}
```

Ниже показаны результаты пробного запуска:

```
Список случайных чисел:
  0.0001    1.6475    2.4332    0.0693    0.7268    0.7383
 24.0357    0.1009   87.1828    5.7361    0.6079    0.6330
  1.6058    0.1406    0.5933    1.1943    5.5295    2.2426
  0.8364    2.7127    0.2514    0.9593    8.9635    0.7139
  0.6249    1.6044    0.8649    2.1577    0.5420   15.0123
  1.7931    1.6183    1.9973    2.9333   12.8512    1.3034
  0.3032    1.1406   18.7880    0.9887

Отсортированный список:
  0.0001    0.0693    0.1009    0.1406    0.2514    0.3032
  0.5420    0.5933    0.6079    0.6249    0.6330    0.7139
  0.7268    0.7383    0.8364    0.8649    0.9593    0.9887
  1.1406    1.1943    1.3034    1.6044    1.6058    1.6183
  1.6475    1.7931    1.9973    2.1577    2.2426    2.4332
  2.7127    2.9333    5.5295    5.7361    8.9635   12.8512
 15.0123   18.7880   24.0357   87.1828
```

Давайте рассмотрим два ключевых момента: использование qsort() и определение mycomp().

Использование функции qsort()

Функция qsort() сортирует массив объектов данных. Ее прототип ANSI имеет следующий вид:

```
void qsort (void *base, size_t nmemb, size_t size,
      int (*compar)(const void *, const void *));
```

Первый аргумент — это указатель на начало сортируемого массива. В программе применяется фактический аргумент vals, представляющий собой имя массива типа double; следовательно, он является указателем на первый элемент массива. В прототипе ANSI для аргумента vals предусмотрено приведение к типу указателя на void. Причина в том, что стандарт ANSI C разрешает приводить любой тип указателя на данные к типу указателя на void, тем самым позволяя первому фактическому аргументу в qsort() ссылаться на массив любого вида.

Во втором аргументе задается количество элементов, предназначенных для сортировки. В листинге 16.17 это N, т.е. число элементов массива. Прототип преобразует это значение в тип size_t.

Третий аргумент — это размер каждого элемента, в данном случае sizeof(double).

Последним аргументом, mycomp, является адрес функции, которая должна использоваться для сравнения элементов.

Определение функции mycomp()

Как упоминалось ранее, прототип qsort() устанавливает форму функции сравнения:

```
int (*compar)(const void *, const void *)
```

Здесь видно, что последний аргумент является указателем на функцию, которая возвращает значение int и принимает два аргумента. Каждый из этих аргументов представляет собой указатель на тип const void. Мы привели в соответствие с ним прототип функции mycomp():

```
int mycomp(const void * p1, const void * p2);
```

Вспомните, что имя функции, передаваемое в качестве аргумента, выступает как указатель на нее, поэтому mycomp совпадает с прототипом compar.

Функция qsort() передает в функцию сравнения адреса двух сравниваемых элементов. В этой программе переменным p1 и p2 присваиваются адреса двух значений типа double, предназначенных для сравнения. Обратите внимание, что первый аргумент в qsort() ссылается на массив в целом, а два аргумента функции сравнения ссылаются на два элемента в массиве. Здесь возникает проблема. Чтобы сравнить значения, для которых доступны только указатели, эти указатели необходимо разыменовать. Так как значения имеют тип double, указатели должны быть разыменованы в тип double. Однако функции qsort() требуются указатели на тип void. Обойти проблему можно, объявив указатели нужного типа внутри функции и инициализировав их значениями, которые передаются в аргументах:

```
/* сортировка по возрастанию */
int mycomp(const void * p1, const void * p2)
{
   /* для доступа к значениям необходимо использовать указатели на double */
   const double * a1 = (const double *) p1;
   const double * a2 = (const double *) p2;

   if (*a1 < *a2)
      return -1;
   else if (*a1 == *a2)
      return 0;
   else
      return 1;
}
```

Короче говоря, в целях универсальности в qsort() и в функции сравнения применяются указатели на void. Как следствие, функции qsort() придется явно сообщить размер каждого элемента массива, а внутри определения функции сравнения преобразовать аргументы типа указателей на void в указатели на подходящий тип данных.

На заметку! void * в C и C++

В языках C и C++ указатели на void трактуются по-разному. В обоих языках вы можете присваивать переменной типа void * указатель любого типа. Например, при вызове функции qsort() в листинге 16.17 выполняется присваивание типа double * указателю на тип void *. Но язык C++ требует приведения типа, когда осуществляется присваивание указателя void * указателю другого типа, в то время как в C такое требование отсутствует. Например, функция mycomp() из листинга 16.17 содержит это приведение типа для указателя p1 типа void *:

```
const double * a1 = (const double *) p1;
```

В языке C подобное приведение типа необязательно; в языке C++ оно обязательно. Поскольку версия с приведением типа работает в обоих языках, имеет смысл использовать его всегда. Впоследствии при переводе программы на язык C++ вам не придется помнить о необходимости изменения этой части кода.

Давайте взглянем на еще один пример функции сравнения. Предположим, что имеются следующие объявления:

```
struct names {
   char first[40];
   char last[40];
};
struct names staff[100];
```

Как должен выглядеть вызов qsort()? Следуя модели, реализованной в листинге 16.17, вызов мог бы иметь следующий вид:

```
qsort(staff, 100, sizeof(struct names), comp);
```

Здесь comp представляет собой имя функции сравнения. На что должна быть похожа эта функция? Пусть необходимо выполнить сортировку по фамилии, а затем по имени. Можно было бы написать следующую функцию:

```
#include <string.h>
int comp(const void * p1, const void * p2)    /* обязательная форма */
{
    /* получение правильного типа указателя */
    const struct names *ps1 = (const struct names *) p1;
    const struct names *ps2 = (const struct names *) p2;
    int res;

    res = strcmp(ps1->last, ps2->last);       /* сравнение фамилий */
    if (res != 0)
        return res;
    else        /* фамилии одинаковы, поэтому сравнить имена */
        return strcmp(ps1->first, ps2->first);
}
```

В данной функции сравнение осуществляется с помощью функции strcmp(), которая возвращает значения, удовлетворяющие требованиям к функции сравнения. Обратите внимание, что для применения операции -> необходим указатель на структуру.

Библиотека утверждений

Библиотека утверждений, поддерживаемая заголовочным файлом assert.h – это небольшая библиотека, предназначенная для оказания содействия при отладке программы. Она состоит из макроса по имени assert(). Макрос принимает в качестве аргумента целочисленное выражение. Если выражение оценивается как ложное (ненулевое), макрос assert() выводит в стандартный поток ошибок (stderr) сообщение об ошибке и вызывает функцию abort(), которая прекращает выполнение программы. (Прототип функции abort() находится в заголовочном файле stdlib.h.) Идея состоит в том, чтобы идентифицировать критические места в программе, где должны быть истинными определенные условия, и с помощью оператора assert() завершать программу, если одно из указанных условий нарушается. Обычно аргументом служит выражение отношения или логическое выражение. Когда assert() прекращает выполнение программы, сначала отображается не прошедший проверку тест, имя файла, содержащего этот тест, и номер строки, где находится тест.

Использование assert()

В листинге 16.18 приведен простой пример применения assert(). В нем утверждается, что значение z должно быть больше или равно 0, прежде чем будет предпринята попытка извлечь из него квадратный корень. Кроме того, ошибочно выполняется вычитание значения вместо его сложения, делая возможным получение переменной z недопустимого значения.

Листинг 16.18. Программа assert.c

```
/* assert.c -- использование assert() */
#include <stdio.h>
#include <math.h>
#include <assert.h>
int main()
{
   double x, y, z;

   puts("Введите пару чисел (0 0 для завершения): ");
   while (scanf("%lf%lf", &x, &y) == 2
               && (x != 0 || y != 0))
   {
      z = x * x - y * y;  /* должно быть + */
      assert(z >= 0);
      printf("результатом является %f\n", sqrt(z));
      puts("Введите следующую пару чисел: ");
   }
   puts("Программа завершена.");

   return 0;
}
```

Ниже показаны результаты пробного запуска:

```
Введите пару чисел (0 0 для завершения):
4 3
результатом является 2.645751
Введите следующую пару чисел:
5 3
результатом является 4.000000
Введите следующую пару чисел:
3 5
Assertion failed: (z >= 0), function main, file /Users/assert.c, line 14.
Отказ утверждения: (z >= 0), функция main, файл /Users/assert.c, строка 14.
```

Точный текст в последней строке зависит от компилятора. Потенциально может сбивать с толку то, что в сообщении не говорится об условии z >= 0; вместо этого в нем уведомляется о том, что отказало утверждение z >= 0.

Чего-то похожего можно было бы добиться с помощью оператора if:

```
if (z < 0)
{
   puts("z меньше 0");
   abort();
}
```

Тем не менее, подход с assert() обладает рядом преимуществ. Он автоматически идентифицирует файл и номер строки, где возникла проблема. Наконец, существует механизм включения и отключения макроса assert() без необходимости в изменении кода. Если вы считаете, что устранили ошибки в программе, поместите следующее определение макроса

```
#define NDEBUG
```

перед местом включения файла assert.h, повторно скомпилируйте программу, и компилятор отключит в файле все операторы assert(). Если проблема возникнет снова, можете удалить директиву #define (или закомментировать ее) и провести повторную компиляцию, в результате чего все операторы assert() снова активизируются.

_Static_assert (C11)

Выражение assert() проверяется во время выполнения. В C11 появилось новое средство в форме объявления _Static_assert, которое осуществляет проверку на этапе компиляции. Таким образом, assert() может привести к прерыванию выполняющейся программы, тогда как _Static_assert() может стать причиной того, что программа не компилируется. Объявление _Static_assert принимает два аргумента. Первым из них является целочисленное константное выражение, а вторым — строка. Если выражение оценивается в 0 (или _False), то компилятор отобразит строку, и не будет компилировать программу. Давайте рассмотрим короткий пример в листинге 16.19, после чего взглянем на отличия между assert() и _Static_assert().

Листинг 16.19. Программа statasrt.c

```
// statasrt.c
#include <stdio.h>
#include <limits.h>
_Static_assert(CHAR_BIT == 16, "Ошибочно предполагается 16-битовый тип char");
int main(void)
{
   puts("Тип char имеет 16 битов.");
   return 0;
}
```

Ниже показана попытка проведения компиляции в командной строке:

```
$ clang statasrt.c
statasrt.c:4:1: error: static_assert failed "Ошибочно предполагается 16-
битовый тип char"
_Static_assert(CHAR_BIT == 16, "Ошибочно предполагается 16-битовый тип char");
^              ~~~~~~~~~~~~~~
1 error generated.
$

statasrt.c:4:1: ошибка: отказ static_assert "Ошибочно предполагается 16-
битовый тип char"
_Static_assert(CHAR_BIT == 16, "Ошибочно предполагается 16-битовый тип char");
^              ~~~~~~~~~~~~~~
1 ошибка сгенерирована.
$
```

Синтаксически _Static_assert трактуется как оператор объявления. Следовательно, в отличие от большинства разновидностей операторов C, он может находиться либо в функции, либо (как в данном случае) быть внешним по отношению к функции.

Требование о том, что первым аргументом в _Static_assert должно быть целочисленное константное выражение, гарантирует возможность его оценки на этапе компиляции. (Вспомните, что выражения sizeof считаются целочисленными константами.) Поэтому в листинге 16.18 вы не можете подставить _Static_assert вместо assert(), т.к. для проверочного выражения в программе используется z > 0, которое является неконстантным и может быть вычислено только во время выполнения. В листинге 16.19 можно было бы применить assert(CHAR_BIT == 16) в теле main(), но это привело бы к выдаче предупреждения об ошибке лишь после компиляции и запуска программы, что менее эффективно.

В заголовочном файле assert.h идентификатор static_assert определен как псевдоним для ключевого слова _Static_assert. Это делает С более совместимым с языком С++, в котором для рассмотренной возможности static_assert используется в качестве ключевого слова.

Функции memcpy() и memmove() из библиотеки string.h

Присваивать один массив другому нельзя, поэтому в таких случаях мы применяли циклы для поэлементного копирования одного массива в другой. Единственное исключение состоит в том, что для символьных массивов мы использовали функции strcpy() и strncpy(). Функции memcpy() и memmove() предлагают почти такие же услуги для других видов массивов. Рассмотрим прототипы этих функций:

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
```

Обе функции копируют n байтов из области, на которую указывает аргумент s2, в область, указанную аргументом s1, и обе они возвращают значение s1. Различие между этими двумя функциями, как указывает ключевое слово restrict, связано с тем, что memcpy() разрешено полагать, что две области памяти нигде не перекрываются друг с другом. Функция memmove() не делает такого предположения, поэтому копирование происходит так, как будто все байты сначала помещаются во временный буфер и только затем копируются в область назначения. Что произойдет, если применить memcpy() к перекрывающимся областям? В этом случае поведение функции не определено, т.е. она может как работать, так и не работать. Компилятор не запрещает использование функции memcpy(), когда этого делать не следует, поэтому именно вы несете ответственность за обеспечение того, что области памяти не перекрываются. Это еще одна часть тяжкой ноши программиста.

Поскольку эти функции предназначены для работы с любым типом данных, два их аргумента имеют тип указателя на void. В С разрешено присваивать указателю типа void* указатель любого типа. Оборотная сторона такой гибкости состоит в том, что функции не способны распознавать, какого типа данные копируются. Поэтому в них присутствует третий аргумент, задающий количество копируемых байтов. Обратите внимание, что для массива количество байтов в общем случае не совпадает с количеством элементов. Таким образом, при копировании массива из 10 значений double в качестве третьего аргумента должно применяться выражение 10*sizeof(double), а не 10.

В листинге 16.20 показаны некоторые примеры использования этих двух функций. В нем предполагается, что тип double имеет в два раза больший размер, чем int, и для проверки этого предположения применяется средство _Static_assert из С11.

Листинг 16.20. Программа mems.c

```
// mems.c -- использование функций memcpy() и memmove()
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define SIZE 10
void show_array(const int ar[], int n);

// удалите следующую строку, если _Static_assert из C11 не поддерживается
_Static_assert(sizeof(double) == 2 * sizeof(int), "double не имеет двойной размер int");
```

```
int main()
{
   int values[SIZE] = {1,2,3,4,5,6,7,8,9,10};
   int target[SIZE];
   double curious[SIZE / 2] = {2.0, 2.0e5, 2.0e10, 2.0e20, 5.0e30};

   puts("Использование memcpy():");
   puts("значения (исходные данные): ");
   show_array(values, SIZE);
   memcpy(target, values, SIZE * sizeof(int));
   puts("целевые данные (копия значений):");
   show_array(target, SIZE);

   puts("\nИспользование memmove() с перекрывающимися областями:");
   memmove(values + 2, values, 5 * sizeof(int));
   puts("значения -- элементы 0-5 скопированы в элементы 2-7:");
   show_array(values, SIZE);

   puts("\nИспользование memcpy() для копирования double в int:");
   memcpy(target, curious, (SIZE / 2) * sizeof(double));
   puts("целевые данные -- 5 значений double в 10 позициях int:");
   show_array(target, SIZE/2);
   show_array(target + 5, SIZE/2);

   return 0;
}

void show_array(const int ar[], int n)
{
   int i;

   for (i = 0; i < n; i++)
      printf("%d ", ar[i]);
   putchar('\n');
}
```

Вот как выглядит вывод:

```
Использование memcpy():
значения (исходные данные):
1 2 3 4 5 6 7 8 9 10
целевые данные (копия значений):
1 2 3 4 5 6 7 8 9 10

Использование memmove() с перекрывающимися областями:
значения -- элементы 0-5 скопированы в элементы 2-7:
1 2 1 2 3 4 5 8 9 10

Использование memcpy() для копирования double в int:
целевые данные -- 5 значений double в 10 позициях int:
0 1073741824 0 1091070464 536870912
1108516959 2025163840 1143320349 -2012696540 1179618799
```

Последний вызов memcpy() копирует данные из массива типа double в массив типа int. Это демонстрирует тот факт, что функция memcpy() ничего не знает, да и не заботится, о типах данных; она просто копирует байты из одной области в другую. (Вы могли бы, к примеру, копировать байты из структуры в массив символов.) Кроме того, никакого преобразования данных не происходит. Если организовать цикл, выполняющий поэлементное присваивание, то значения типа double будут преобразованы в тип int. В этом случае байты копируются в том виде, как есть, и программа затем интерпретирует комбинации битов, как если бы они имели тип int.

Переменное число аргументов: файл stdarg.h

Ранее в этой главе обсуждались макросы с переменным числом аргументов. Заголовочный файл stdarg.h предоставляет похожую возможность для функций. Однако использовать ее немного сложнее. Вы должны выполнить следующие действия.

1. Подготовить прототип функции, в котором применяется троеточие.
2. Создать в определении функции переменную типа va_list.
3. Использовать макрос для инициализации этой переменной списком аргументов.
4. Применить макрос для доступа к списку аргументов.
5. Использовать макрос для очистки.

Давайте рассмотрим эти действия более подробно. Прототип для функции подобного рода должен иметь список, содержащий, по крайней мере, один параметр, за которым следует троеточие:

```
void f1(int n, ...);                    // допустимо
int f2(const char * s, int k, ...);     // допустимо
char f3(char c1, ..., char c2);         // недопустимо, троеточие не в конце
double f3(...);                         // недопустимо, параметры отсутствуют
```

Крайний справа параметр (предшествующий троеточию) играет специальную роль; для его обозначения в стандарте используется термин *parmN*. В предшествующих примерах роль parmN играл параметр n в первом случае и k — во втором. Фактическим аргументом, передаваемым этому параметру, является количество аргументов, которые представлены разделом троеточия. Например, прототипированную ранее функцию f1() можно вызывать следующим образом:

```
f1(2, 200, 400);          // 2 дополнительных аргумента
f1(4, 13, 117, 18, 23);   // 4 дополнительных аргумента
```

Тип va_list, объявленный в заголовочном файле stdargs.h, представляет объект данных, применяемый для хранения параметров, которые соответствуют разделу троеточия в списке параметров. Начало определения функции с переменным числом аргументов выглядит примерно так:

```
double sum(int lim,...)
{
  va_list ap;              // объявление объекта для хранения аргументов
```

В этом примере lim является параметром parmN и указывает количество аргументов в списке переменных-аргументов.

Затем функция будет использовать макрос va_start(), также определенный в stdargs.h, для копирования списка аргументов в переменную va_list. Макрос принимает два аргумента: переменную va_list и параметр parmN. Продолжая предыдущий пример, переменная va_list названа ap, а параметру parmN назначено имя lim, так что вызов будет иметь следующий вид:

```
va_start(ap, lim);       // инициализация ap списком аргументов
```

На следующем этапе производится доступ к содержимому списка аргументов. Это предусматривает применение еще одного макроса, va_arg(), который принимает два аргумента: переменную типа va_list и имя типа. При первом вызове он возвращает первый элемент списка, при следующем вызове — следующий элемент списка и т.д.

Например, если первым аргументом в списке был double, а вторым — int, вы могли бы поступить так:

```
double tic;
int toc;
...
tic = va_arg(ap, double);      // извлечение первого аргумента
toc = va_arg(ap, int);         // извлечение второго аргумента
```

Но будьте внимательны. Тип аргумента на самом деле должен соответствовать спецификации. Если первым аргументом является 10.0, предыдущий код для tic работает нормально. Однако если аргументом оказывается 10, код может не заработать; автоматическое преобразование double в int, предпринимаемое для операции присваивания, здесь не происходит.

Наконец, вы должны провести очистку с помощью макроса va_end(). Например, может понадобиться освободить память, динамически выделенную для хранения аргументов. Этот макрос принимает в качестве аргумента переменную va_list:

```
va_end(ap);                    // очистка
```

После этого переменная ap может оказаться непригодной к употреблению до тех пор, пока вы не инициализируете ее повторно посредством макроса va_start.

Поскольку макрос va_arg() не обеспечивает копирование предыдущих аргументов для их возможного восстановления, может оказаться целесообразным сохранение копии переменной va_list. Для этой цели в стандарте C99 предусмотрен макрос по имени va_copy(). Он принимает два аргумента типа va_list и копирует второй аргумент в первый:

```
va_list ap;
va_list apcopy;
double
double tic;
int toc;
...
va_start(ap, lim);             // инициализация ap списком аргументов
va_copy(apcopy, ap);           // делает apcopy копией ap
tic = va_arg(ap, double);      // извлечение первого аргумента
toc = va_arg(ap, int);         // извлечение второго аргумента
```

На данном этапе по-прежнему можно извлечь первые два элемента из apcopy, несмотря на то, что они были удалены из ap.

В листинге 16.21 приведен краткий пример использования этих возможностей для создания функции, которая суммирует переменное число аргументов; здесь первым аргументом sum() является количество суммируемых элементов.

Листинг 16.21. Программа varargs.c

```
//varargs.c -- использование переменного числа аргументов
#include <stdio.h>
#include <stdarg.h>
double sum(int, ...);

int main(void)
{
   double s,t;

   s = sum(3, 1.1, 2.5, 13.3);
   t = sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1);
```

```
    printf("возвращаемое значение "
          "sum(3, 1.1, 2.5, 13.3):                %g\n", s);
    printf("возвращаемое значение "
          "sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1): %g\n", t);
    return 0;
}
double sum(int lim,...)
{
    va_list ap;                   // объявление объекта для хранения аргументов
    double tot = 0;
    int i;
    va_start(ap, lim);            // инициализация ap списком аргументов
    for (i = 0; i < lim; i++)
      tot += va_arg(ap, double);  // доступ к каждому элементу в списке аргументов
    va_end(ap);                   // очистка
    return tot;
}
```

Ниже показаны результаты пробного запуска:

```
возвращаемое значение sum(3, 1.1, 2.5, 13.3):                16.9
возвращаемое значение sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1): 31.6
```

Проверив вычисления, вы обнаружите, что функция sum() сложила три числа при первом вызове и шесть чисел – при втором.

В общем случае, функции с переменным числом аргументов сложнее в использовании по сравнению с макросами такого рода, но имеют более широкий диапазон применений.

Ключевые понятия

Стандарт С не просто описывает язык С; он определяет пакет, состоящий из языка С, препроцессора С и стандартной библиотеки С. Препроцессор позволяет выполнить подготовительные действия перед компиляцией, указывая необходимые подстановки, выбирая строки кода, подлежащие компиляции, а также устанавливая другие аспекты поведения компилятора. Библиотека С расширяет возможности языка и предоставляет готовые решения для многих задач программирования.

Резюме

Препроцессор С и библиотека С представляют собой два важных дополнения языка С. Препроцессор С, следуя специальным директивам, нужным образом подстраивает исходный код перед его компиляцией. Библиотека С предоставляет множество функций, предназначенных для содействия в решении таких задач, как ввод, вывод, операции с файлами, управление памятью, сортировка и поиск, математические вычисления, обработка строк и множество других. В разделе V приложения Б содержится полный список функций библиотеки ANSI C.

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля приведены в приложении А.

1. Ниже приведены группы из одного или нескольких макросов, сопровождаемые строками кода, в которых они используются. Каким будет результат выполнения кода в каждом случае? Является ли код допустимым? (Предполагается, что переменные были объявлены.)

   а.
   ```
   #define FPM 5280    /* футов в миле */
   dist = FPM * miles;
   ```
   б.
   ```
   #define FEET 4
   #define POD FEET + FEET
   plort  = FEET * POD;
   ```
   в.
   ```
   #define SIX = 6;
   nex = SIX;
   ```
   г.
   ```
   #define NEW(X) X + 5
   y = NEW(y);
   berg = NEW(berg) * lob;
   est = NEW(berg) / NEW(y);
   nilp = lob * NEW(-berg);
   ```

2. Исправьте определение в части г) вопроса 1, чтобы сделать код более надежным.

3. Определите функциональный макрос, который возвращает меньшее из двух значений.

4. Определите макрос `EVEN_GT(X,Y)`, который возвращает значение 1, если `X` является четным, а также больше `Y`.

5. Определите функциональный макрос, который выводит представления и значения двух целочисленных выражений. Например, он может выводить строку

   ```
   3 + 4 = 7 и 4 * 12 = 48
   ```

   если аргументами являются выражения `3 + 4` и `4 * 12`.

6. Напишите операторы `#define` для достижения следующих целей.

   а. Создайте именованную константу со значением `25`.

   б. Обеспечьте, чтобы идентификатор `SPACE` представлял символ пробела.

   в. Обеспечьте, чтобы макрос `PS()` выводил символ пробела.

   г. Обеспечьте, чтобы макрос `BIG(X)` представлял сложение `3` и `X`.

   д. Обеспечьте, чтобы макрос `SUMSQ(X,Y)` представлял сумму квадратов `X` и `Y`.

7. Определите макрос, который выводит имя, значение и адрес переменной `int` в следующем формате:

   ```
   имя: fop;   значение: 23;   адрес: ff46016
   ```

8. Предположим, что имеется блок кода, который необходимо пропустить во время тестирования программы. Как это сделать без удаления этого блока кода из файла?

9. Напишите фрагмент кода, который выводит дату обработки препроцессором, если макрос `PR_DATE` определен.

10. При обсуждении встраиваемых функций были показаны три разных версии функции square(). Чем они отличаются друг от друга в плане поведения?

11. Создайте макрос, используя выражение обобщенного выбора, которое оценивается в строку "boolean", если аргумент макроса имеет тип _Bool, и в строку "not boolean" в противном случае.

12. Что неправильно в следующей программе?

```
#include <stdio.h>
int main(int argc, char argv[])
{
   printf("Квадратный корень из %f равен %f\n", argv[1],
         sqrt(argv[1]) );
}
```

13. Предположим, что scores — это массив из 1000 значений int, которые требуют сортировки в порядке убывания, а также, что вы используете функцию сортировки qsort() и функцию сравнения по имени comp().

    а. Как правильно вызвать qsort()?

    б. Какое определение подойдет для comp()?

14. Предположим, что data1 — это массив из 100 значений double, а data2 — массив из 300 значений double.

    а. Напишите вызов функции memcpy(), который скопирует первые 100 элементов data2 в data1.

    б. Напишите вызов функции memcpy(), который скопирует последние 100 элементов data2 в data1.

Упражнения по программированию

1. Начните разработку заголовочного файла с определениями препроцессора, которые вы хотите использовать.

2. Гармоническое среднее двух чисел получается путем вычисления среднего от инверсий этих чисел с последующим инвертированием результата. Воспользуйтесь директивой #define для определения функционального макроса, который выполняет эту операцию. Напишите простую программу для тестирования этого макроса.

3. В полярной системе координат вектор описывается модулем и углом с осью *x* в направлении против часовой стрелки. В прямоугольной системе координат тот же вектор описывается составляющими *x* и *y* (рис. 16.3). Напишите программу, которая считывает значения модуля и угла (в градусах) вектора, а затем отображает составляющие *x* и *y*. Воспользуйтесь следующими уравнениями:

   ```
   x = r cos A    y = r sin A
   ```

   Для выполнения преобразования применяйте функцию, которая принимает структуру, содержащую полярные координаты, и возвращает структуру, содержащую прямоугольные координаты (или, если хотите, выберите вариант с указателями на эти структуры).

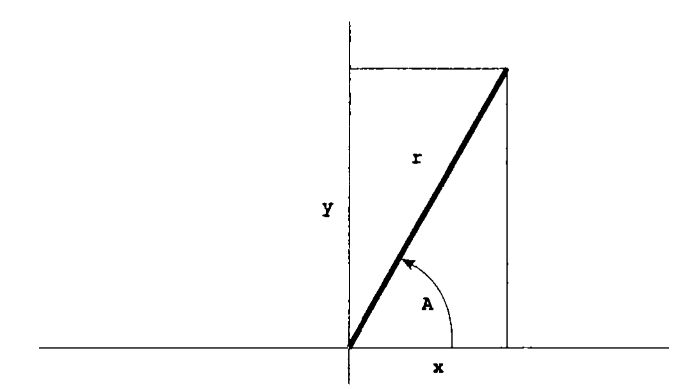


Рис. 16.3. Прямоугольные и полярные координаты

4. Библиотека ANSI содержит функцию `clock()` со следующим описанием:

```
#include <time.h>
clock_t clock (void);
```

Здесь `clock_t` – тип данных, определенный в файле `time.h`. Функция возвращает процессорное время, выраженное в единицах, которые зависят от реализации. (Если процессорное время недоступно или не может быть представлено, функция возвращает `-1`.) Однако в файле `time.h` также определена константа `CLOCKS_PER_SEC`, которая представляет количество единиц процессорного времени в секунде. Следовательно, в результате деления разницы между двумя значениями, возвращаемыми `clock()`, на константу `CLOCKS_PER_SEC` получается количество секунд, прошедшее между двумя вызовами функции. Приведение значений к типу `double` до операции деления позволит получить результат в долях секунды. Напишите функцию, которая принимает аргумент типа `double`, представляющий промежуток времени, а затем выполняет цикл до истечения указанного периода времени. Напишите простую программу для тестирования этой функции.

5. Напишите функцию, которая в качестве аргумента принимает имя массива элементов `int`, размер массива и значение, представляющее количество выборок. Функция должна случайным образом выбирать из массива указанное количество элементов и выводить их значения. Ни один элемент массива не должен выбираться более одного раза. (Это эмулирует выбор чисел в лотерее или членов жюри.) Если в данной реализации доступна функция `time()` (которая обсуждалась в главе 12) или подобная ей функция, то для вывода данных воспользуйтесь функцией `srand()`, чтобы инициализировать генератор случайных чисел `rand()`. Напишите простую программу для тестирования этой функции.

6. Модифицируйте код в листинге 16.15 так, чтобы программа использовала массив элементов `struct names` (как определено после листинга) вместо массива элементов `double`. Задействуйте меньше элементов и явно инициализируйте массив подходящим набором имен.

7. Ниже приведена часть программы, использующей функцию с переменным числом аргументов:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
void show_array(const double ar[], int n);
double * new_d_array(int n, ...);
int main()
{
  double * p1;
  double * p2;

  p1 = new_d_array(5, 1.2, 2.3, 3.4, 4.5, 5.6);
  p2 = new_d_array(4, 100.0, 20.00, 8.08, -1890.0);
  show_array(p1, 5);
  show_array(p2, 4);
  free(p1);
  free(p2);

  return 0;
}
```

Функция new_d_array() принимает аргумент типа int и переменное количество аргументов double. Она возвращает указатель на блок памяти, выделенный функцией malloc(). Аргумент int задает количество элементов, которые должны быть в динамическом массиве, а значения double предназначены для инициализации элементов, при этом первое значение присваивается первому элементу, второе – второму и т.д. Завершите программу, предоставив код для функций show_array() и new_d_array().

17

Расширенное представление данных

В ЭТОЙ ГЛАВЕ...

- Функции: дополнительные сведения о функции malloc ()
- Использование С для представления разнообразных типов данных
- Новые алгоритмы и увеличение возможностей концептуальной разработки программ
- Абстрактные типы данных

Изучение языка программирования подобно обучению музыке, плотницкому делу или инженерному искусству. Вначале вы знакомитесь с инструментами и средствами измерений, учитесь держать в руках молоток и избегать ударов по пальцам, а также решать бесчисленные проблемы, связанные с падением, соскальзыванием и утерей равновесия различных объектов. До сих пор в процессе чтения этой книги вы приобретали теоретические и практические навыки в создании переменных, структур, функций и тому подобного. Однако со временем вы переходите на более высокий уровень, на котором навыки использования инструментов превращаются во вторую натуру, а реальной задачей становится проектирование и реализация проекта. Постепенно у вас вырабатывается способность восприятия проекта как единого целого.

Данная глава как раз и посвящена этому более высокому уровню работы. Изложенный в ней материал может показаться более сложным для восприятия, чем материал, изложенный в предшествующих главах, однако его усвоение может оказаться и более плодотворным, поскольку позволяет ученику стать мастером.

Мы начнем с ознакомления с чрезвычайно важным аспектом проектирования программы: способом представления данных. Зачастую наиболее важным аспектом разработки программы является выбор подходящего представления данных, которыми будет манипулировать эта программа. Правильный выбор представления данных может превратить написание остальной программы в очень простую задачу. Вы уже знакомы с встроенными типами данных С: простыми переменными, массивами, указателями, структурами и объединениями.

Тем не менее, часто выбор правильного представления данных не ограничивается простым выбором типа. Вы должны также подумать и о том, какие операции придется выполнять. То есть потребуется выбрать способ хранения данных и определить, какие операции допустимы для такого типа данных. Например, в реализациях С тип `int` и тип указателя обычно хранятся как целые числа, но для каждого из них определен свой набор допустимых операций. Скажем, одно целое число можно умножить на другое, но нельзя умножать указатель на указатель. Операцию * можно применять для разыменования указателя, но она бессмысленна для целочисленного значения.

В языке С определены допустимые операции для его фундаментальных типов. Тем не менее, при проектировании схемы представления данных может понадобиться определить допустимые операции самостоятельно. На языке С это можно делать путем разработки функций, представляющих желаемые операции. Короче говоря, проектирование типа данных состоит из определения способа хранения данных и разработки функций для управления данными.

Вы также ознакомитесь с некоторыми *алгоритмами* — готовыми рецептами для манипулирования данными. Как программист, вы со временем обзаведетесь набором таких рецептов, которые будете снова и снова применять для решения похожих задач.

В этой главе рассматривается процесс проектирования типов данных — процесс сопоставления алгоритмов с представлениями данных. Здесь вы столкнетесь с рядом распространенных форм данных, таких как очередь, список и двоичное дерево поиска.

В главе будет также представлена концепция абстрактного типа данных (abstract data type — ADT). Тип ADT упаковывает методы и представления данных проблемно-ориентированным, а не языково-ориентированным способом. После того как вы спроектировали тип ADT, его можно легко многократно использовать при различных обстоятельствах. Понимание типов ADT концептуально подготовит вас к вступлению в мир объектно-ориентированного программирования и языка C++.

Исследование представления данных

Давайте начнем с обдумывания данных. Предположим, что требуется создать программу для адресной книги. Какую форму данных необходимо использовать для хранения информации? Поскольку с каждой записью связана разнообразная информация, каждую запись имеет смысл представить в виде структуры. А как представить несколько записей? С помощью стандартного массива структур? Посредством динамического массива? С помощью какой-то другой формы? Должны ли записи быть упорядочены в алфавитном порядке? Требуется ли возможность поиска в записях по почтовому индексу? Нужен ли поиск по междугородному телефонному коду? Действия, которые требуется выполнять, могут влиять на выбор способа хранения информации. Короче говоря, прежде чем приступать к созданию кода, придется принять массу проектных решений.

А как вы представите растровые графические изображения, которые должны храниться в памяти? В растровом изображении каждый пиксель на экране устанавливается индивидуально. Во времена черно-белых экранов для представления одного пикселя можно было использовать один бит (1 или 0) – отсюда и английское название растровых графических изображений *bitmapped* (побитовое отображение). На цветных мониторах описание одного пикселя занимает более одного бита. Например, выделение по 8 бит каждому пикселю позволяет получить 256 цветов. В настоящее время произошел переход к 65 536 цветам (16 бит на пиксель), 16 777 216 цветам (24 бита на пиксель), 2 147 483 648 (32 бита на пиксель) и даже больше. При наличии 32-битовых цветов и разрешающей способности монитора 2560×1440 пикселей для представления одного экрана растровой графики вам понадобится около 118 миллионов битов (14 Мбайт). Следует ли смириться с этим или же разработать какой-то метод сжатия информации? Должно ли это сжатие выполняться *без потерь* или *с потерями* (сравнительно неважных данных)? И снова, прежде чем погружаться в кодирование, придется принять множество проектных решений.

Рассмотрим конкретный случай представления данных. Предположим, что нужно написать программу, которая позволяет вводить список всех фильмов (на видеокассетах, дисках DVD и дисках Blu-ray), просмотренных в течение года. Для каждого фильма желательно регистрировать разнообразную информацию, такую как название, год выпуска, имена и фамилии режиссера и ведущих актеров, продолжительность и жанр (комедия, научная фантастика, романтика, мелодрама и т.п.), рейтинг и т.д. Это предполагает применение структуры для каждого фильма и массива структур для списка фильмов. В целях простоты ограничим структуру двумя членами: названием фильма и собственной оценкой его рейтинга по 10-бальной шкале. В листинге 17.1 приведена элементарная реализация, использующая этот подход.

Листинг 17.1. Программа `films1.c`

```
/* films1.c -- использование массива структур */
#include <stdio.h>
#include <string.h>
#define TSIZE      45   /* размер массива для хранения названия      */
#define FMAX       5    /* максимальное количество названий фильмов */

struct film {
   char title[TSIZE];
   int rating;
};
char * s_gets(char * st, int n);
```

```
int main(void)
{
   struct film movies[FMAX];
   int i = 0;
   int j;

   puts("Введите название первого фильма:");
   while (i < FMAX && s_gets(movies[i].title, TSIZE) != NULL &&
        movies[i].title[0] != '\0')
   {
     puts("Введите свое значение рейтинга <0-10>:");
     scanf("%d", &movies[i++].rating);
     while(getchar() != '\n')
        continue;
     puts("Введите название следующего фильма (или пустую строку для прекращения ввода):");
   }
   if (i == 0)
      printf("Данные не введены.");
   else
      printf("Список фильмов:\n");

   for (j = 0; j < i; j++)
      printf("Фильм: %s  Рейтинг: %d\n", movies[j].title,
            movies[j].rating);
   printf("Программа завершена.\n");

   return 0;
}

char * s_gets(char * st, int n)
{
   char * ret_val;
   char * find;

   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      find = strchr(st, '\n');    // поиск новой строки
      if (find)                   // если адрес не равен NULL,
         *find = '\0';            // поместить туда нулевой символ
      else
         while (getchar() != '\n')
            continue;             // отбросить остаток строки
   }
   return ret_val;
}
```

Программа создает массив структур и заполняет его данными, которые вводит пользователь. Ввод продолжается вплоть до заполнения массива (проверка FMAX), до достижения конца файла (проверка NULL) или до нажатия пользователем клавиши <Enter> в начале строки (проверка '\0').

Такая организация программы сопряжена с рядом проблем. Во-первых, скорее всего, программа будет напрасно тратить большой объем памяти, поскольку названия большинства фильмов содержат меньше 40 символов, но, в то же время, названия некоторых фильмов могут быть весьма длинными, такими как "Скромное обаяние буржуазии" или "Вон Тон Тон, пес, который спас Голливуд". Во-вторых, ограничение в пять фильмов в год многим покажется излишне строгим. Конечно, этот предел можно увеличить, но каким он должен быть? Кто-то просматривает до 500 фильмов в год, поэтому значение FMAX можно было бы увеличить до 500, но для некоторых и этого может

оказаться слишком мало, в то время как для других оно приводило бы к напрасной трате огромного объема памяти. Кроме того, некоторые компиляторы по умолчанию ограничивают объем памяти, доступной для переменных с автоматическим классом хранения наподобие movies, и такой крупный массив мог бы превысить указанное ограничение. Ситуацию можно исправить, сделав массив статическим или внешним либо проинструктировав компилятор о необходимости применения стека большего размера. Однако это не решает действительную проблему.

Действительная проблема здесь заключается в том, что представление данных определено совершенно негибким образом. На этапе компиляции вам приходится принимать решения, которые целесообразнее принимать во время выполнения. Это предполагает переход к представлению данных, которое использует динамическое выделение памяти. Можно попробовать следующий код:

```
#define TSIZE  45          /* размер массива для хранения названия  */
struct film {
    char title[TSIZE];
    int rating;
};
...
  int n, i;
  struct film * movies;   /* указатель на структуру */
  ...
  printf("Укажите максимальное количество фильмов, которые вы будете вводить:\n");
  scanf("%d", &n);
  movies = (struct film *) malloc(n * sizeof(struct film));
```

Здесь, как и в главе 12, указатель movies можно применять так, как если бы он был именем массива:

```
while (i < FMAX && gets(movies[i].title) != NULL &&
    movies[i].title[0] != '\0')
```

За счет использования функции malloc() вы можете отложить определение количества элементов до момента выполнения программы, поэтому не придется выделять память для 500 элементов, если их необходимо только 20. Но при таком подходе обязанность ввести корректное значение для количества записей возлагается на пользователя.

От массива к связному списку

В идеале было бы желательно иметь возможность добавлять данные неограниченно (или до тех пор, пока программа не исчерпает свою доступную память), не указывая заранее количество записей, которые будут созданы, и не вынуждая программу выделять огромное пространство памяти без реальной на то необходимости. Этой цели можно достигнуть, вызывая malloc() после ввода каждой записи и выделяя лишь такой объем памяти, которого достаточно для новой записи. Если пользователь вводит информацию о трех фильмах, программа вызывает функцию malloc() три раза. Если пользователь вводит информацию о 300 фильмах, программа вызывает malloc() триста раз.

Такая, прекрасная на первый взгляд, идея порождает новую проблему. Чтобы увидеть, в чем она заключается, сравните однократный вызов malloc() для выделения памяти под 300 структур film с 300-кратным вызовом этой функции для выделения памяти каждый раз только для одной структуры film. В первом случае память распределяется в виде одного непрерывного блока, и для отслеживания содержимого требуется

единственная переменная указателя на структуру (`film`), которая указывает на первую структуру в блоке. Как было показано в приведенном ранее фрагменте кода, простая форма записи с массивом обеспечивает указателю доступ к каждой структуре внутри блока. Проблема со вторым подходом – отсутствие какой-либо гарантии того, что последовательные вызовы `malloc()` приведут к выделению смежных блоков памяти. Это означает, что структуры не обязательно будут сохранены непрерывно (рис. 17.1). Таким образом, вместо хранения одного указателя на блок из 300 структур придется хранить 300 указателей – по одному для каждой независимо выделенной структуры!

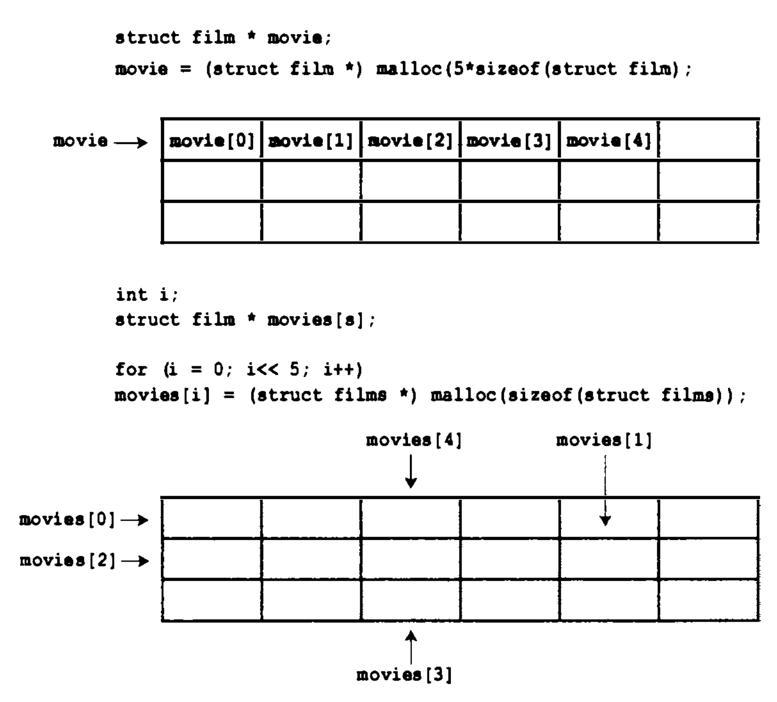


Рис. 17.1. Выделение памяти под структуры одним блоком и выделение памяти под структуры индивидуально

Одно из возможных решений, которое, однако, мы применять не будем, предполагает создание большого массива указателей и присваивание значений указателям друг за другом по мере выделения памяти под новые структуры:

```
#define TSIZE  45           /* размер массива для хранения названий      */
#define FMAX   500          /* максимальное количество названий фильмов */
struct film {
   char title[TSIZE];
   int rating;
};
...
struct film * movies[FMAX]; /* массив указателей на структуры */
int i;
...
movies[i] = (struct film *) malloc (sizeof (struct film));
```

Этот подход позволяет сэкономить большой объем памяти, если вы не используете полный комплект указателей, т.к. массив из 500 указателей занимает значительно меньше памяти, чем массив из 500 структур. Тем не менее, по-прежнему пространс-

тво тратится впустую на хранение неиспользуемых указателей, к тому же продолжает действовать ограничение в 500 структур.

Существует более эффективный способ. При каждом вызове функции malloc() для выделения памяти под новую структуру одновременно можно выделять память и для нового указателя. Но вы можете возразить, что тогда потребуется еще один указатель для отслеживания вновь выделенного указателя, а для его отслеживания необходим еще один указатель, и так до бесконечности. Предотвратить эту потенциальную проблему можно путем переопределения структуры так, чтобы она включала указатель на *следующую* структуру. Тогда при каждом создании новой структуры ее адрес можно будет сохранять в предыдущей структуре. Короче говоря, структуру film нужно переопределить так, как показано ниже:

```
#define TSIZE  45     /* размер массива для хранения названий  */
struct film {
   char title[TSIZE];
   int rating;
   struct film * next;
};
```

Действительно, структура не может содержать структуру того же самого типа, но может иметь указатель на структуру такого же типа. Определение подобного рода служит основой *связного списка* – списка, в котором каждый элемент содержит информацию о местонахождении следующего элемента.

Прежде чем взглянуть на код С для связного списка, давайте подробнее рассмотрим концепции, лежащие в основе такого списка. Предположим, что в качестве названия фильма пользователь вводит Modern Times и 10 для значения рейтинга. Программа выделила бы память для структуры film, скопировала бы строку Modern Times в член title и установила бы значение члена rating равным 10. Чтобы указать на то, что за этой структурой никаких других структур не следует, программа должна была бы установить значение члена-указателя next в NULL. (Вспомните, что NULL – символическая константа, определенная в файле stdio.h, которая представляет нулевой указатель.) Разумеется, необходимо отслеживать место хранения первой структуры. Это можно делать, присвоив адрес отдельному указателю, который мы будем называть *указателем на заголовок списка*. Указатель на заголовок указывает на первый элемент в связном списке элементов. На рис. 17.2 показано, как выглядит эта структура. (Пустая область в члене title сжата для уменьшения размера рисунка.) Теперь предположим, что пользователь вводит название и рейтинг второго фильма – скажем, Midnight in Paris и 8. Программа выделяет память для второй структуры film и сохраняет адрес новой структуры в члене next первой структуры (перезаписывая ранее установленное значение NULL), чтобы указатель next ссылался на следующую структуру в связном списке.

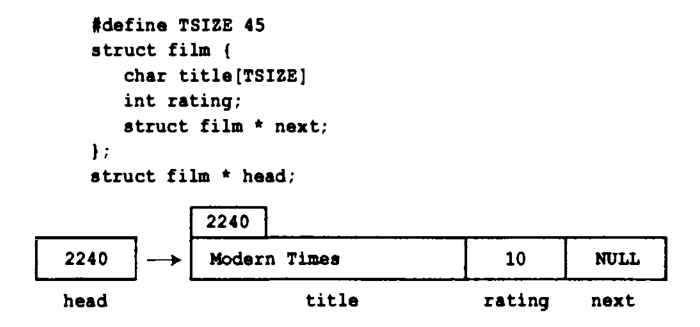


Рис. 17.2. Первый элемент в связном списке

Затем программа копирует значения `Midnight in Paris` и `8` в новую структуру и устанавливает значение ее члена `next` в `NULL`, указывая, что теперь эта структура является последней в списке. Такой список из двух элементов продемонстрирован на рис. 17.3.

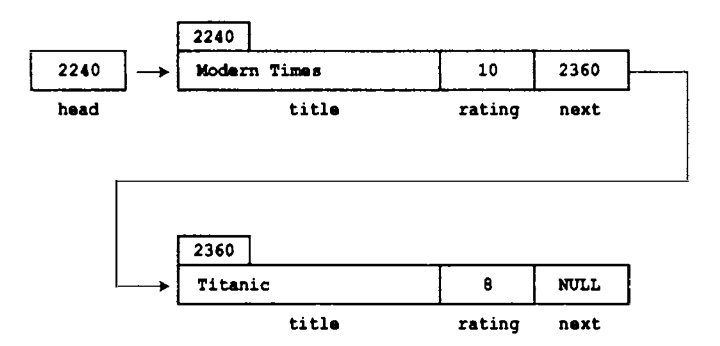


Рис. 17.3. Связный список с двумя элементами

Обработка информации о каждом новом фильме будет выполняться аналогично. Адрес новой структуры будет сохраняться в предыдущей структуре, в новую структуру будет помещаться введенная информация, а значение члена `next` новой структуры будет устанавливаться в `NULL`, что приведет к созданию связного списка, подобного представленному на рис. 17.4.

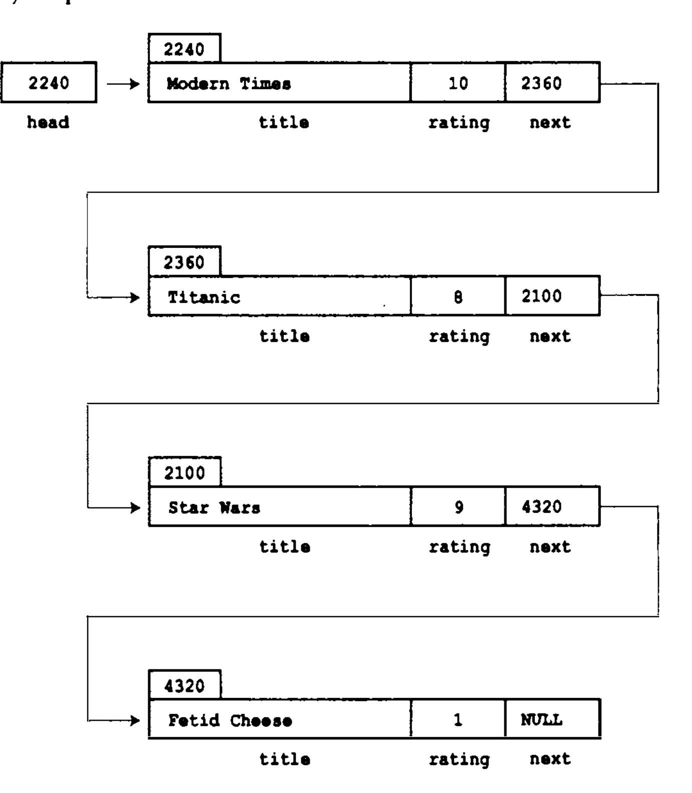


Рис. 17.4. Связный список с несколькими элементами

Предположим, что список необходимо отобразить. При каждом выводе элемента для нахождения следующего отображаемого элемента можно применять адрес, сохраненный в соответствующей структуре. Однако чтобы эта схема работала, необходим указатель, который будет отслеживать самый первый элемент в списке, т.к. ни одна структура в списке не хранит адрес первого элемента. К счастью, это уже сделано с помощью указателя на заголовок списка.

Использование связного списка

Теперь, когда вы получили представление о работе связного списка, давайте реализуем его. В листинге 17.2 представлен модифицированный код из листинга 17.1, в котором для хранения информации о фильмах вместо массива применяется связный список.

Листинг 17.2. Программа `films2.c`

```
/* films2.c -- использование связного списка структур */
#include <stdio.h>
#include <stdlib.h>      /* содержит прототип функции malloc()   */
#include <string.h>      /* содержит прототип функции strcpy()   */
#define TSIZE   45       /* размер массива для хранения названия */

struct film {
   char title[TSIZE];
   int rating;
   struct film * next;   /* указывает на следующую структуру в списке */
};
char * s_gets(char * st, int n);

int main(void)
{
   struct film * head = NULL;
   struct film * prev, * current;
   char input[TSIZE];

   /* Сбор и сохранение информации */
   puts("Введите название первого фильма:");
   while (s_gets(input, TSIZE) != NULL && input[0] != '\0')
   {
     current = (struct film *) malloc(sizeof(struct film));
     if (head == NULL)       /* первая структура     */
        head = current;
     else                    /* последующие структуры */
        prev->next = current;
     current->next = NULL;
     strcpy(current->title, input);
     puts("Введите свое значение рейтинга <0-10>:");
     scanf("%d", &current->rating);
     while(getchar() != '\n')
        continue;
     puts("Введите название следующего фильма (или пустую строку для прекращения ввода):");
       prev = current;
   }

   /* Отображение списка фильмов */
   if (head == NULL)
      printf("Данные не введены.");
   else
      printf("Список фильмов:\n");
   current = head;
```

```
    while (current != NULL)
    {
        printf("Фильм: %s  Рейтинг: %d\n",
               current->title, current->rating);
        current = current->next;
    }
    /* Программа выполнена, поэтому можно освободить память */
    current = head;
    while (current != NULL)
    {
        current = head;
        head = current->next;
        free(current);
    }
    printf("Программа завершена.\n");

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');      // поиск новой строки
        if (find)                     // если адрес не равен NULL,
            *find = '\0';             // поместить туда нулевой символ
        else
            while (getchar() != '\n')
                continue;             // отбросить остаток строки
    }
    return ret_val;
}
```

Программа решает две задачи с использованием связного списка. Во-первых, она конструирует список и заполняет его входными данными. Во-вторых, она отображает список. Отображение списка — более простая задача, поэтому вначале рассмотрим ее.

Отображение списка

Идея заключается в том, чтобы начать с установки указателя (назовем его current) в ссылку на первую структуру. Поскольку указатель на заголовок (по имени head) уже указывает, куда нужно, следующего кода вполне достаточно:

```
current = head;
```

Затем с помощью формы записи с указателем можно обратиться к членам этой структуры:

```
printf("Фильм: %s  Рейтинг: %d\n", current->title, current->rating);
```

Далее указатель current переустанавливается для ссылки на следующую структуру в списке. Эта информация хранится в члене next структуры, поэтому задача решается посредством такого кода:

```
current = current->next;
```

По завершении весь процесс необходимо повторить. После отображения последнего элемента в списке указатель current будет установлен в NULL, т.к. это значение члена next последней структуры. Данным обстоятельством можно воспользоваться для прекращения вывода. Фрагмент кода из films2.c, применяемый для отображения списка, выглядит следующим образом:

```
while (current != NULL)
{
   printf("Фильм: %s  Рейтинг: %d\n", current->title, current->rating);
   current = current->next;
}
```

Почему бы для перемещения по списку не воспользоваться head вместо того, чтобы создавать новый указатель (current)? Причина в том, что это привело бы к изменению значения head, и программа лишилась бы возможности находить начало списка.

Создание списка

Создание списка предусматривает выполнение трех действий.

1. Использование функции malloc() для выделения достаточного пространства под структуру.
2. Сохранение адреса структуры.
3. Копирование в структуру корректной информации.

Не имеет смысла создавать структуру, если она пока не требуется, поэтому для приема от пользователя информации о названии фильма в программе применяется временное хранилище (массив input). Если пользователь эмулирует с помощью клавиатуры символ EOF или вводит пустую строку, цикл ввода завершается:

```
while (s_gets(input, TSIZE) != NULL && input[0] != '\0')
```

При наличии введенных данных программа запрашивает пространство для структуры и присваивает ее адрес переменной типа указателя current:

```
current = (struct film *) malloc(sizeof(struct film));
```

Адрес самой первой структуры должен быть сохранен в переменной типа указателя head. Адрес каждой последующей структуры должен сохраняться в члене next предыдущей структуры. Таким образом, программе необходим способ для выяснения того, является ли текущая структура первой. Проще всего решить эту задачу, инициализировав указатель head значением NULL в начале программы. Затем в программе можно использовать значение указателя head для принятия решения о дальнейших действиях:

```
if (head == NULL)       /* первая структура      */
   head = current;
else                    /* последующие структуры */
   prev->next = current;
```

В этом коде prev — указатель на структуру, выделенную в прошлый раз.

Далее понадобится установить члены структуры в соответствующие значения. В частности, член next должен быть установлен в NULL для указания на то, что текущая структура является последней в списке. Вы должны скопировать название фильма из массива input в член title и получить значение для члена rating. Эти действия выполняет следующий код:

```
current->next = NULL;
strcpy(current->title, input);
puts("Введите свое значение рейтинга <0-10>:");
scanf("%d", &current->rating);
```

Поскольку вызов s_gets() ограничивает вводимые данные пределом в TSIZE - 1 символов, строка в массиве input поместится в член title, поэтому вполне безопасно применять функцию strcpy().

Наконец, вы должны подготовить программу к следующему циклу ввода. В частности, указатель prev необходимо установить так, чтобы он ссылался на текущую структуру, т.к. после ввода названия следующего фильма и распределения следующей структуры текущая структура станет предыдущей. Программа устанавливает этот указатель в конце цикла:

```
prev = current;
```

Работает ли программа? Ниже показаны результаты пробного запуска.

```
Введите название первого фильма:
Spirited Away
Введите свое значение рейтинга <0-10>:
9
Введите название следующего фильма (или пустую строку для прекращения ввода):
The Duelists
Введите свое значение рейтинга <0-10>:
8
Введите название следующего фильма (или пустую строку для прекращения ввода):
Devil Dog: The Mound of Hound
Введите свое значение рейтинга <0-10>:
1
Введите название следующего фильма (или пустую строку для прекращения ввода):

Список фильмов:
Фильм: Spirited Away  Рейтинг: 9
Фильм: The Duelists  Рейтинг: 8
Фильм: Devil Dog: The Mound of Hound  Рейтинг: 1
Программа завершена.
```

Освобождение памяти, занимаемой списком

Во многих средах программа освободит память, выделенную с помощью функции malloc(), при своем завершении, но лучше, чтобы в привычку вошло уравновешивание каждого вызова malloc() вызовом free(). Таким образом, программа очищает используемую память с применением функции free() к каждой выделенной структуре:

```
current = head;
while (current != NULL)
{
    free(current);
    current = current->next;
}
```

Дополнительные соображения

Возможности программы films2.c несколько ограничены. Например, в ней отсутствует проверка, удалось ли функции malloc() найти запрошенную память, и она лишена каких-либо средств для удаления элементов из списка. Тем не менее, такие упущения могут быть устранены. Скажем, можно добавить код, который проверяет, является ли возвращаемое значение malloc() равным NULL (признак неудачи в полу-

чении желаемой памяти). Если программа нуждается в удалении записей, в ней можно предусмотреть дополнительный код.

Такой специализированный подход к решению проблем и добавлению функциональных возможностей по мере необходимости не всегда является наилучшим стилем программирования. С другой стороны, обычно не удается предугадать абсолютно все, что потребуется программе. По мере роста масштаба проектов модель заблаговременного планирования всех необходимых функциональных средств становится все менее реалистичной. Было замечено, что самыми успешными оказывались те крупные программы, которые поэтапно развивались от небольших удачных программ.

Учитывая, что планы могут пересматриваться, имеет смысл разрабатывать первоначальные идеи в манере, упрощающей модификацию. Пример в листинге 17.2 не следует этому принципу. Так, в нем проявляется тенденция к смешиванию деталей кодирования и концептуальной модели. Например, в этом коде концептуальная модель заключается в том, что элементы добавляются в список. Программа затеняет этот интерфейс, вынося на передний план такие детали, как `malloc()` и указатель `current->next`. Весьма желательно, если бы вы смогли писать программу в стиле, который делает очевидным то, что вы добавляете элемент в список, и одновременно скрывает такие вспомогательные действия, как вызов функций управления памятью и установка указателей. Отделение пользовательского интерфейса от деталей реализации упростит понимание и обновление программы. Упомянутых целей можно достичь, создав программу заново. Начав разработку с нуля, вы можете достичь таких целей. Давайте посмотрим как.

Абстрактные типы данных

В программировании вы пытаетесь сопоставить необходимый тип данных с нуждами программной задачи. Например, для представления количества имеющихся пар обуви можно было бы использовать тип `int`, а для представления средней цены одной пары — тип `float` или `double`. В приведенных примерах программ, связанных с фильмами, данные формировали список элементов, каждый из которых состоял из названия фильма (строки С) и значения рейтинга (типа `int`). Ни один из базовых типов С не соответствует этому описанию, поэтому для представления отдельных элементов мы определили структуру, а затем создали пару методов для объединения последовательности структур в список. В сущности, мы применили возможности языка С по разработке нового типа данных, удовлетворяющего конкретным потребностям, но делали это бессистемно. Теперь мы примем систематичный подход к определению типов.

Что образует тип? *Тип* определяют два вида информации: набор свойств и набор операций. Например, свойство типа `int` заключается в том, что он представляет целочисленное значение и, следовательно, разделяет свойства целых значений. Разрешенными арифметическими операциями для этого типа являются изменение знака, сложение двух значений `int`, вычитание одного значения `int` из другого, умножение двух значений `int`, деление одного значения `int` на другое и получение результата вычисления одного значения `int` по модулю другого. Объявление переменной типа `int` означает, что на нее могут воздействовать эти и только эти операции.

НА ЗАМЕТКУ! Свойства целочисленного типа

В основе типа `int` языка С лежит более абстрактная концепция *целого числа*. Математики определяют свойства целых чисел в формальной абстрактной манере. Например, если N и M — целые числа, то $N + M = M + N$, или для любых двух целых чисел N и M существует целое число S, такое что $N + M = S$. Если $N + M = S$ и $N + Q = S$, то $M = Q$.

Можно считать, что математика предлагает абстрактную концепцию целого числа, а язык С — реализацию этой концепции. Например, в С предоставляются средства хранения целого числа и выполнения целочисленных операций, таких как сложение и умножение. Обратите внимание, что обеспечение поддержки арифметических операций является важной частью представления целых чисел. Тип `int` был бы значительно менее полезным, если бы он позволял только хранить значения, но не использовать их в арифметических выражениях. Также следует отметить, что задача представления целых чисел в этой реализации решена далеко не идеально. Например, существует бесконечное количество целых чисел, но 2-байтовый тип `int` может представлять только 65 536 из них; не путайте абстрактную идею с конкретной реализацией.

Предположим, что вы хотите определить новый тип данных. Во-первых, вы должны предоставить способ для хранения данных — возможно, за счет проектирования структуры. Во-вторых, понадобится обеспечить методы для манипулирования данными. В качестве примера рассмотрим программу `films2.c` (листинг 17.2). Она содержит связанный набор структур для хранения информации, а также код для добавления и отображения информации. Тем не менее, программа не решает эти задачи так, чтобы сделать очевидным создание нового типа данных. Как же следовало поступить?

Науки о вычислениях предлагают очень эффективный способ определения новых типов данных. Он является трехэтапным процессом перехода от абстрактного к конкретному.

1. Предоставьте абстрактное описание свойств типа и операций, которые можно выполнять над этим типом. Такое описание не должно быть привязано ни к какой конкретной реализации. Оно даже не должно быть привязано к конкретному языку программирования. Формальное абстрактное описание подобного рода называют *абстрактным типом данных* (abstract data type — ADT).

2. Разработайте программный интерфейс, реализующий этот тип ADT. То есть укажите, как следует хранить данные, и опишите набор функций, которые выполняют желаемые операции. Например, в С вы можете предоставить определение структуры наряду с прототипами функций для манипулирования структурами. Эти функции играют для определенного пользователем типа ту же самую роль, которую встроенные операции С исполняют для фундаментальных типов С. Любой, кто захочет воспользоваться новым типом, будет применять этот интерфейс в своих программах.

3. Напишите код для реализации интерфейса. Конечно, этот шаг очень важен, но программисту, который использует новый тип, совершенно не обязательно знать подробности реализации.

Чтобы посмотреть, как работает этот процесс, давайте рассмотрим конкретный пример. Поскольку мы уже приложили кое-какие усилия к примеру с созданием списка фильмов, переделаем его с применением нового подхода.

Получение абстракции

По существу все, что требуется для проекта информации о фильмах — это список элементов. Каждый элемент содержит название и рейтинг фильма. Нам необходимо иметь возможность добавления новых элементов в конец списка и отображения его содержимого. Давайте назовем абстрактный тип, который будет удовлетворять этим потребностям, *списком*. Какими свойствами он должен обладать? Понятно, что список должен уметь сохранять последовательность элементов. Другими словами, список может содержать несколько элементов, причем эти элементы каким-то образом упоря-

дочены, что позволяет говорить о первом, втором или последнем элементе в списке. Далее, тип списка должен поддерживать такие операции, как добавление элемента в список. Ниже перечислены некоторые полезные операции:

- инициализация списка пустым содержимым;
- добавление элемента в конец списка;
- определение, является ли список пустым;
- определение, является ли список полным;
- определение количества элементов в списке;
- посещение каждого элемента в списке с целью выполнения какого-то действия, такого как отображение элемента.

Для этого проекта дополнительные операции не нужны, но более универсальный перечень операций со списками может включать следующие:

- вставка элемента в любое место списка;
- удаление элемента из списка;
- извлечение элемента из списка (список остается неизмененным);
- замена одного элемента в списке другим;
- поиск элемента в списке.

Тогда неформальное, но абстрактное определение списка выглядит так: список — это объект данных, способный хранить последовательность элементов, к которому можно применять любые из перечисленных ранее операций. В этом определении не заявлен вид элементов, которые могут храниться в списке. В нем не указано, должен ли для хранения элементов использоваться массив, связанный набор структур либо иная форма данных. Определение не диктует, какой метод применять, например, для выяснения количества элементов в списке. Все эти детали оставлены за реализацией.

Для простоты давайте примем в качестве абстрактного типа данных упрощенный список, содержащий только те функциональные возможности, которые требуются для проекта информации о фильмах. Краткое описание этого типа приведено ниже.

Имя типа:	Простой список
Свойства типа:	Может содержать последовательность элементов
Операции типа:	Инициализация списка пустым содержимым Определение, является ли список пустым Определение, является ли список полным Определение количества элементов в списке Добавление элемента в конец списка Обход списка с обработкой каждого элемента Опустошение списка

Следующим этапом является разработка для ADT простого списка интерфейса на языке C.

Построение интерфейса

Интерфейс для простого списка состоит из двух частей. Первая часть описывает способ представления данных, а вторая — функции, реализующие операции ADT. Например, интерфейс будет содержать функции для добавления элемента в список и

вывода количества элементов в списке. Проектное решение интерфейса должно как можно ближе отражать описание ADT. Следовательно, оно должно быть выражено в терминах некоторого общего типа Item, а не в терминах какого-то конкретного типа вроде int или struct film. Один из способов достижения этого предполагает использование средства typedef языка C для определения Item в качестве требуемого типа:

```
#define TSIZE  45     /* размер массива для хранения названия */
struct film
{
   char title[TSIZE];
   int rating;
};
typedef struct film Item;
```

Затем тип Item можно применять в остальных определениях. Если позже потребуется список элементов какой-то другой формы данных, можно будет переопределить тип Item и оставить остальную часть определения интерфейса без изменений.

После того как тип Item определен, необходимо принять решение о способе хранения элементов этого типа. В действительности этот шаг относится к этапу реализации, но принятие решения в настоящий момент упростит понимание примера. Подход с использованием связанных структур достаточно успешно работал в программе films2.c, поэтому применим его, как показано ниже:

```
typedef struct node
{
   Item item;
   struct node * next;
} Node;
typedef Node * List;
```

В реализации с применением связного списка каждая связь называется *узлом*. Каждый узел содержит информацию, формирующую содержимое списка, и указатель на следующий узел. Чтобы подчеркнуть используемую терминологию, мы назвали структуру узла избитым именем node (т.е. "узел") и применили typedef, чтобы сделать Node именем типа для структуры struct node. Наконец, для управления связным списком необходим указатель на его начало, поэтому мы использовали typedef, чтобы превратить List в имя для указателя этого типа. Таким образом, объявление

```
List movies;
```

устанавливает movies как указатель, подходящий для ссылки на связный список.

Является ли этот способ определения типа List единственным? Нет. Например, для отслеживания количества записей можно было бы задействовать переменную:

```
typedef struct list
{
   Node * head;      /* указатель на заголовок списка      */
   int size;         /* количество записей в списке        */
} List;              /* альтернативное определение списка */
```

Можно было бы добавить второй указатель, предназначенный для отслеживания конца списка. Позже вы увидите соответствующий пример. Пока давайте ограничимся первым определением типа List. Важно помнить, что объявление

```
List movies;
```

следует рассматривать как определение списка, а не установку указателя на узел или структуры.

Точное представление данных списка movies является деталью реализации, которая не должна быть видна на уровне интерфейса.

Например, при запуске программа должна инициализировать указатель на заголовок значением NULL, но не следует применять код вроде такого:

```
movies = NULL;
```

А почему? По той причине, что впоследствии может оказаться, что реализация типа List в виде структуры подходит больше, и тогда потребуется следующая инициализация:

```
movies.next = NULL;
movies.size = 0;
```

Никто из тех, кто использует тип List, не должен беспокоиться о подобных нюансах. Вместо этого должна быть возможность записывать приблизительно такой код:

```
InitializeList(movies);
```

Программистам требуется знать только о том, что для инициализации списка они должны применять функцию InitializeList(). Они не обязаны знать точную реализацию данных для переменной List. Это является примером *сокрытия данных* – искусства маскировки подробностей представления данных от более высоких уровней программирования.

Для предоставления руководства пользователю прототип функции можно сопровождать следующими строками:

```
/* операция:      инициализация списка                       */
/* предусловия:  plist указывает на список list              */
/* постусловия:  список инициализирован пустым содержимым   */
void InitializeList(List * plist);
```

Есть три момента, на которые вы должны обратить внимание. Во-первых, комментарии описывают *предусловия*, т.е. условия, которые должны быть удовлетворены до вызова функции. Например, здесь необходим список, предназначенный для инициализации. Во-вторых, комментарии описывают *постусловия* – условия, которые должны быть удовлетворены после выполнения функции. Наконец, в-третьих, в качестве своего аргумента функция использует указатель на список, а не сам список, поэтому вызов функции будет иметь такой вид:

```
InitializeList(&movies);
```

Причина заключается в том, что в С аргументы передаются по значению. Таким образом, единственный способ позволить функции С изменять значения из вызывающей программы предусматривает применение указателя на эту переменную. Как видите, здесь ограничения языка приводят к некоторому отличию интерфейса от его абстрактного описания.

Принятый в языке С метод объединения информации о типе и функциях в единый пакет предполагает помещение определений для типа и прототипов функций (в том числе комментариев с пред- и постусловиями) в заголовочный файл. Этот файл должен предоставлять всю информацию, в которой нуждается программист для использования типа. Заголовочный файл для простого типа list показан в листинге 17.3. В нем конкретная структура определена как относящаяся к типу Item, после чего тип Node определен в терминах Item и тип List – в терминах Node. Затем в функциях, представляющих операции над списком, типы Item и List применяются для аргументов. Если функции необходимо модифицировать аргумент, она использует указатель

на соответствующий тип, а не сам тип напрямую. В файле имена функций начинаются с прописных букв для их обозначения как части интерфейсного пакета. Кроме того, для защиты от множественного включения файла применяется прием с `#ifndef`, который обсуждался в главе 16. Если ваш компилятор не поддерживает тип `bool` из стандарта C99, можете заменить в заголовочном файле строку

```
#include <stdbool.h>   /* функциональная возможность C99 */
```

такой строкой:

```
enum bool {false, true}; /*определение bool как типа, и false, true как значений*/
```

Листинг 17.3. Заголовочный файл для интерфейса `list.h`

```
/* list.h -- заголовочный файл для простого типа списка */
#ifndef LIST_H_
#define LIST_H_
#include <stdbool.h>    /* функциональная возможность C99 */

/* объявления, специфичные для программы */

#define TSIZE     45   /* размер массива для хранения названия */
struct film
{
   char title[TSIZE];
   int rating;
};
/* определения общих типов */

typedef struct film Item;

typedef struct node
{
   Item item;
   struct node * next;
} Node;
typedef Node * List;

/* прототипы функций */

/* операция:     инициализация списка                              */
/* предусловия:  plist указывает на список                         */
/* постусловия:  список инициализирован пустым содержимым          */
void InitializeList(List * plist);

/* операция:    определение, является ли список пустым             */
/*               plist указывает на инициализированный список      */
/* постусловия:  функция возвращает значение True, если список     */
/*               пуст, и False в противном случае                  */
bool ListIsEmpty(const List *plist);

/* операция:    определение, является ли список полным             */
/*               plist указывает на инициализированный список      */
/* постусловия:  функция возвращает значение True, если список     */
/*            полон, и False в противном случае                    */
bool ListIsFull(const List *plist);

/* операция:    определение количества элементов в списке          */
/*               plist указывает на инициализированный список      */
/* постусловия: функция возвращает число элементов в списке        */
unsigned int ListItemCount(const List *plist);

/* операция:     добавление элемента в конец списка                */
/* предусловия:  item — элемент, добавляемый в список              */
/*               plist указывает на инициализированный список      */
```

```
/* постусловия:  если возможно, функция добавляет элемент в       */
/*               конец списка и возвращает значение True;         */
/*               в противном случае возвращается значение False   */
bool AddItem(Item item, List * plist);

/* операция:    применение функции к каждому элементу списка      */
/*               plist указывает на инициализированный список     */
/*               pfun указывает на функцию, которая принимает     */
/*               аргумент Item и не имеет возвращаемого значения  */
/* постусловия:  функция, указанная pfun, выполняется один        */
/*               раз для каждого элемента в списке                */
void Traverse (const List *plist, void (* pfun)(Item item) );

/* операция:    освобождение выделенной памяти, если она есть     */
/*               plist указывает на инициализированный список     */
/* постусловия:  любая память, выделенная для списка,             */
/*               освобождается, и список устанавливается          */
/*               в пустое состояние                               */
void EmptyTheList(List * plist);

#endif
```

Список модифицируют только функции InitializeList(), AddItem() и EmptyTheList, поэтому формально только они требуют аргумента типа указателя. Однако если бы пользователю пришлось помнить о необходимости передачи аргумента List одним функциям и его адреса другим, то это могло бы приводить к путанице. Таким образом, для упрощения задачи пользователя во всех функциях используются аргументы типа указателей.

Один из прототипов в заголовочном файле несколько сложнее остальных:

```
/* операция:     применение функции к каждому элементу списка      */
/* предусловия:  plist указывает на инициализированный список      */
/*               pfun указывает на функцию, которая принимает      */
/*               аргумент Item и не имеет возвращаемого значения   */
/* постусловия:  функция, указанная pfun, выполняется один         */
/*               раз для каждого элемента в списке                 */
void Traverse (const List *plist, void (* pfun)(Item item) );
```

Аргумент pfun представляет собой указатель на функцию. В этом случае он является указателем на функцию, которая принимает значение item в качестве аргумента и не имеет возвращаемого значения. Возможно, вы помните из главы 14, что указатель на функцию можно передавать в виде аргумента другой функции, которая сможет вызывать эту указанную функцию. Так, например, pfun может указывать на функцию, отображающую элемент. Функция Traverse() будет применять эту функцию к каждому элементу списка, в результате отображая весь список.

Использование интерфейса

Мы заявляем, что этот интерфейс можно использовать для написания программы, не располагая никакими дополнительными деталями – например, ничего не зная о том, как реализованы функции интерфейса. Давайте прямо сейчас напишем новую версию программы вывода информации о фильмах еще до создания вспомогательных функций. Поскольку интерфейс определен в терминах типов List и Item, программа должна быть создана с применением этих же типов. Ниже показан один из возможных планов, представленный с помощью псевдокода:

```
Создать переменную List.
Создать переменную Item.
Инициализировать список пустым содержимым.
Пока список не заполнен и есть входные данные:
   Прочитать входные данные и поместить их в переменную Item.
   Добавить элемент в конец списка.
Посетить каждый элемент списка и отобразить его.
```

Программа, приведенная в листинге 17.4, следует этому базовому плану; кроме того, в нее добавлен код для проверки ошибок. Взгляните, как в ней используется интерфейс, описанный в файле `list.h` (листинг 17.3). Обратите также внимание, что листинг содержит код функции `showmovies()`, которая соответствует прототипу, требуемому функцией `Traverse()`. Поэтому программа может передавать указатель `showmovies` в функцию `Traverse()`, чтобы та могла применять функцию `showmovies()` к каждому элементу списка. (Вспомните, что имя функции является указателем на эту функцию.)

Листинг 17.4. Программа films3.c

```
/* films3.c -- использование связного списка в стиле ADT */
/* компилировать вместе с list.c                         */
#include <stdio.h>
#include <stdlib.h>    /* прототип для exit()    */
#include "list.h"      /* определение List, Item */
void showmovies(Item item);
char * s_gets(char * st, int n);
int main(void)
{
   List movies;
   Item temp;

   /* инициализация */
   InitializeList(&movies);
   if (ListIsFull(&movies))
   {
      fprintf(stderr, "Доступная память отсутствует! Программа завершена.\n");
      exit(1);
   }

   /* сбор и сохранение информации */
   puts("Введите название первого фильма:");
   while (s_gets(temp.title, TSIZE) != NULL && temp.title[0] != '\0')
   {
     puts("Введите свое значение рейтинга <0-10>:");
     scanf("%d", &temp.rating);
     while(getchar() != '\n')
          continue;
     if (AddItem(temp, &movies)==false)
     {
          fprintf(stderr, "Проблема с выделением памяти\n");
          break;
     }
     if (ListIsFull(&movies))
     {
          puts("Список полон.");
          break;
     }
     puts("Введите название следующего фильма (или пустую строку для прекращения ввода):");
   }
```

```
    /* отображение */
    if (ListIsEmpty(&movies))
        printf("Данные не введены.");
    else
    {
        printf("Список фильмов:\n");
        Traverse(&movies, showmovies);
    }
    printf("Вы ввели %d фильмов.\n", ListItemCount(&movies));

    /* очистка */
    EmptyTheList(&movies);
    printf("Программа завершена.\n");

    return 0;
}

void showmovies(Item item)
{
    printf("Фильм: %s  Рейтинг: %d\n", item.title,
            item.rating);
}

char * s_gets(char * st, int n)

    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');    // поиск новой строки
        if (find)                   // если адрес не равен NULL,
            *find = '\0';           // поместить туда нулевой символ
        else
            while (getchar() != '\n')
                continue;           // отбросить остаток строки
    }
    return ret_val;
}
```

Реализация интерфейса

Разумеется, еще предстоит реализовать интерфейс List. Подход, принятый в C, предусматривает сбор определений функций в файле по имени list.c. Тогда полная программа будет состоять из трех файлов: list.h, в котором определены структуры данных и предоставлены прототипы для пользовательского интерфейса, list.c, содержащего код функций для реализации интерфейса, и films3.c, представляющего собой файл исходного кода, где интерфейс списка применяется для решения конкретной задачи. Одна из возможных реализаций файла list.c показана в листинге 17.5. Чтобы запустить программу, необходимо скомпилировать оба файла films3.c и list.c и скомпоновать их. (Компиляция многофайловых программ обсуждалась в главе 9). Вместе файлы list.c, list.c и films3.c образуют завершенную программу (рис. 17.5).

Листинг 17.5. Файл реализации list.c

```
/* list.c -- функции для поддержки операций со списком */
#include <stdio.h>
#include <stdlib.h>
#include "list.h"
```

```
/* прототип локальной функции */
static void CopyToNode(Item item, Node * pnode);

/* функции интерфейса */
/* устанавливает список в пустое состояние */
void InitializeList(List * plist)
{
   * plist = NULL;
}

/* возвращает true, если список пуст */
bool ListIsEmpty(const List * plist)
{
   if (*plist == NULL)
      return true;
   else
      return false;
}

/* возвращает true, если список полон */
bool ListIsFull(const List * plist)
{
   Node * pt;
   bool full;

   pt = (Node *) malloc(sizeof(Node));
   if (pt == NULL)
      full = true;
   else
      full = false;
   free(pt);

   return full;
}

/* возвращает количество узлов */
unsigned int ListItemCount(const List * plist)
{
   unsigned int count = 0;
   Node * pnode = *plist;    /* установка в начало списка */

   while (pnode != NULL)
   {
      ++count;
      pnode = pnode->next;  /* установка в следующий узел */
   }

   return count;
}

/* создает узел для хранения элемента и добавляет его в конец  */
/* списка, указанного переменной plist (медленная реализация) */
bool AddItem(Item item, List * plist)
{
   Node * pnew;
   Node * scan = *plist;

   pnew = (Node *) malloc(sizeof(Node));
   if (pnew == NULL)
      return false;       /* выход из функции в случае ошибки */

   CopyToNode(item, pnew);
   pnew->next = NULL;
   if (scan == NULL)         /* список пуст, поэтому поместить */
      *plist = pnew;        /* pnew в начало списка            */
```

```
    else
    {
        while (scan->next != NULL)
            scan = scan->next;   /* поиск конца списка     */
        scan->next = pnew;       /* добавление pnew в конец */
    }

    return true;
}

/* посещает каждый узел и выполняет функцию, указанную pfun */
void Traverse  (const List * plist, void (* pfun)(Item item) )
{
    Node * pnode = *plist;       /* установка в начало списка */

    while (pnode != NULL)
    {
        (*pfun)(pnode->item);   /* применение функции к элементу */
        pnode = pnode->next;    /* переход к следующему элементу */
    }
}

/* освобождает память, выделенную функцией malloc() */
/* устанавливает указатель списка в NULL          */
void EmptyTheList(List * plist)
{
    Node * psave;

    while (*plist != NULL)
    {
        psave = (*plist)->next; /* сохранение адреса следующего узла */
        free(*plist);           /* освобождение текущего узла       */
        *plist = psave;         /* переход к следующему узлу        */
    }
}

/* определение локальной функции */
/* копирует элемент в узел      */
static void CopyToNode(Item item, Node * pnode)
{
    pnode->item = item;         /* копирование структуры */
}
```

Замечания по поводу программы

С файлом list.c связано много интересных особенностей. Скажем, он иллюстрирует ситуацию, когда можно использовать функции с внутренним связыванием. Как описано в главе 12, функции с внутренним связыванием известны только в файле, где они определены. При реализации интерфейса иногда удобно применять вспомогательные функции, которые не являются частью официального интерфейса. Например, в приведенной программе функция CopyToNode() используется для копирования значения типа Item в переменную типа Item. Поскольку эта функция — часть реализации, но не интерфейса, с помощью квалификатора класса хранения static мы скрыли ее в файле list.c. А теперь давайте проанализируем остальные функции.

Функция InitializeList() инициализирует список пустым содержимым. В нашей реализации это означает установку переменной типа List в NULL. Как упоминалось ранее, это требует передачи в функцию указателя на переменную типа List.

Функция ListEmpty() довольно проста, но она полагается на то, переменная списка установлена в NULL, когда список пуст. Таким образом, важно инициализировать список до первого вызова функции ListEmpty().

```
list.h

/* list.h--заголовочный файл для простого типа списка */
/* объявления, специфичные для программы */
#define TSIZE 45 /* размер массива для хранения названия */
struct film
{
  char title[TSIZE];
  int rating;
};
.
.
.
void Traverse (List l, void (* pfun)(Item item) );
```

```
list.c

/* list.c--функции для поддержки операций со списком */
#include<stdio.h>
#include<stdlib.h>
#include "list.h"

.
.
.
/* копирует элемент в узел */
static void CopyToNode (Item item, Node * pnode)
{
  pnode->item = item; /* копирование структуры */
}
```

```
films3.c

/* films3.c -- использование связного списка в стиле ADT */
#include <stdio.h>
#include <stdlib.h> /* прототип для exit() */
#include "list.h"
void showmovies(Item item);

int main(void)
{
.
.
.
}
```

Рис. 17.5. *Три части программного пакета*

Кроме того, если вы расширите интерфейс, включив в него средство для удаления элементов, то должны удостовериться, что функция удаления сбрасывает список в пустое состояние после удаления последнего элемента. В случае применения связного списка его размер ограничен объемом доступной памяти. Функция `ListIsFull()` пытается выделить объем памяти, достаточный для нового элемента. Если это ей не удается, значит, список полон. Если попытка была успешной, функция должна освободить только что выделенную память, чтобы она была доступна для реального элемента.

Функция `ListItemCount()` использует обычный алгоритм обхода связного списка, подсчитывая при этом количество элементов:

```
unsigned int ListItemCount(const List * plist)
{
   unsigned int count = 0;
   Node * pnode = *plist;    /* установка в начало списка */
   while (pnode != NULL)
   {
      ++count;
```

```
      pnode = pnode->next;  /* установка в следующий узел */
   }
   return count;
}
```

Функция `AddItem()` наиболее сложная из всех:

```
bool AddItem(Item item, List * plist)
{
   Node * pnew;
   Node * scan = *plist;
   pnew = (Node *) malloc(sizeof(Node));
   if (pnew == NULL)
      return false;          /* выход из функции в случае ошибки */
   CopyToNode(item, pnew);
   pnew->next = NULL;
   if (scan == NULL)         /* список пуст, поэтому поместить   */
      *plist = pnew;         /* pnew в начало списка             */
   else
   {
      while (scan->next != NULL)
         scan = scan->next;  /* поиск конца списка               */
      scan->next = pnew;     /* добавление pnew в конец          */
   }
   return true;
}
```

Первым делом функция `AddItem()` выделяет память для нового узла. Если это ей удается, она применяет функцию `CopyToNode()` для копирования элемента в узел. Затем она устанавливает член `next` узла в `NULL`. Как вы помните, это служит сигналом того, что данный узел является последним в связном списке. И, наконец, после создания узла и присваивания соответствующих значений его членам функция присоединяет узел в конец списка. Если это первый добавленный элемент списка, программа устанавливает указатель на заголовок в первый элемент. (Вспомните, что функция `AddItem()` вызывается с адресом указателя на заголовок во втором аргументе, поэтому `* plist` — это значение указателя на заголовок.) В противном случае код выполняет проход по связному списку до тех пор, пока не обнаружит элемент, член `next` которого установлен в `NULL`. В текущий момент этот узел является последним в списке, поэтому функция переустанавливает его член `next`, чтобы он указывал на новый узел.

Принятая практика программирования требует вызова функции `ListIsFull()` перед попыткой добавления элемента в список. Однако пользователь может упустить этот момент, поэтому функция `AddItem()` самостоятельно проверяет успешность вызова `malloc()`. Кроме того, вполне вероятно, что между вызовами функций `ListIsFull()` и `AddItem()` пользователь мог выполнить еще какие-то действия по выделению памяти, поэтому лучше на всякий случай проверить, сработала ли функция `malloc()`.

Функция `Traverse()` аналогична `ListItemCount()`, но в ней добавлено применение функции к каждому элементу списка:

```
void Traverse  (const List * plist, void (* pfun)(Item item) )
{
   Node * pnode = *plist;      /* установка в начало списка      */
   while (pnode != NULL)
   {
      (*pfun)(pnode->item);    /* применение функции к элементу */
      pnode = pnode->next;     /* переход к следующему элементу */
   }
}
```

Вспомните, что pnode->item представляет данные, хранящиеся в узле, а pnode->next идентифицирует следующий узел в связном списке. Например, вызов

```
Traverse(movies, showmovies);
```

применяет функцию showmovies() к каждому элементу в списке.

Наконец, функция EmptyTheList() освобождает память, которая ранее была выделена с помощью malloc():

```
void EmptyTheList(List * plist)
{
   Node * psave;
   while (*plist != NULL)
   {
      psave = (*plist)->next; /* сохранение адреса следующего узла  */
      free(*plist);           /* освобождение текущего узла         */
      *plist = psave;         /* переход к следующему узлу          */
   }
}
```

В этой реализации пустой список обозначается путем установки переменной List в NULL. Следовательно, чтобы можно было изменять переменную List, в функцию должен быть передан адрес этой переменной. Так как List уже является указателем, то plist — это указатель на указатель. Таким образом, внутри кода выражение *plist имеет тип указателя на Node. Когда список заканчивается, значение *plist равно NULL, т.е. исходный фактический аргумент теперь установлен в NULL.

Код сохраняет адрес следующего узла, поскольку в принципе вызов функции free() может сделать содержимое текущего узла (на который ссылается указатель *plist) более недоступным.

НА ЗАМЕТКУ! Ограничения const

Некоторые функции обработки списка имеют в качестве параметра выражение List * plist. Это отражает тот факт, что такие функции не модифицируют список. Здесь const обеспечивает определенную защиту, предотвращая изменение указателя *plist (величины, на которую указывает plist). В рассматриваемой программе plist указывает на movies, так что спецификатор const предотвращает изменение этими функциями переменной movies, которая, в свою очередь, указывает на первую ссылку в списке. Таким образом, код вроде показанного ниже недопустим, скажем, в функции ListItemCount():

```
*plist = (*plist)->next;      // не разрешено, если *plist - константа
```

Это хорошо, поскольку изменение *plist и, следовательно, movies привело бы утере программой возможности отслеживания данных. Однако то, что переменные *plist и movies трактуются как const, совершенно не означает, что данные, на которые указывает *plist или movies, являются константами. Например, следующий код вполне допустим:

```
(*plist)->item.rating = 3;    // разрешено, даже если *plist - константа
```

Причина в том, что этот код не изменяет переменную *plist; он изменяет данные, на которые указывает *plist. Вывод из всего сказанного заключается в том, что на const нельзя полностью полагаться при выявлении программных ошибок, которые приводят к случайному изменению данных.

Анализ проделанной работы

Сейчас мы посвятим некоторое время оценке того, что нам дал подход с использованием ADT. Для начала сравним листинги 17.2 и 17.4. В обеих программах для решения задачи с созданием списка фильмов применяется один и тот же фундаментальный

метод (динамическое выделение памяти для связанных структур). Но программа в листинге 17.2 показывает все программные нюансы, помещая malloc() и prev->next в открытое представление. С другой стороны, код в листинге 17.4 скрывает эти детали и выражает программу на языке, который напрямую связан с решаемой задачей. Это значит, что в нем речь идет о создании списка и добавлении в него элементов, а не о вызове функций управления памятью или о переустановке указателей. Короче говоря, листинг 17.4 представляет программу в терминах решаемой задачи, а не в терминах низкоуровневых инструментов, необходимых для ее решения. Версия с ADT ориентирована на проблемы конечного пользователя, поэтому читать ее гораздо легче.

Вместе файлы list.h и list.c образуют многократно используемый ресурс. Если вам необходим простой список элементов другого типа, достаточно обратиться к этим файлам. Предположим, что необходимо хранить сведения о своих родственниках: имена, родственные отношения, адреса и номера телефонов. Прежде всего, следует обратиться к файлу list.h и переопределить тип Item:

```
typedef struct itemtag
{
   char fname[14];
   char lname [24];
   char relationship[36];
   char address [60];
   char phonenum[20];
} Item;
```

Затем... что ж, в данном случае это все, что вы должны были сделать, поскольку все функции простого списка определены в терминах типа Item. В некоторых случаях пришлось бы также переопределить функцию CopyToNode(). Например, если бы элемент был массивом, то его не удалось бы копировать с помощью операции присваивания.

Еще один важный момент связан с тем, что пользовательский интерфейс определен в терминах операций абстрактного списка, а не какого-то конкретного набора представлений данных и алгоритмов. Это позволяет свободно манипулировать реализацией, не переделывая конечную программу. Например, созданная нами функция AddItem() несколько неэффективна, т.к. она всегда начинает работу с начала списка и затем выполняет поиск его конца. Указанный недостаток можно устранить, отслеживая конец списка. Например, тип List можно переопределить следующим образом:

```
typedef struct list
{
   Node * head;     /* указывает на начало списка*/
   Node * end;      /* указывает на конец списка  */
} List;
```

Конечно, после этого пришлось бы переписать функции обработки списка, применив это новое определение, но не нужно было бы изменять что-либо в листинге 17.4. Такой вид изолирования реализации от финального интерфейса особенно полезен в крупномасштабных программных проектах. Этот подход называется *сокрытием данных*, т.к. подробное представление данных скрыто от конечного пользователя.

Обратите внимание, что этот конкретный тип ADT даже не требует реализации простого списка в виде связного списка. Ниже показана еще одна возможность:

```
#define MAXSIZE 100
typedef struct list
{
   Item entries[MAXSIZE];     /* массив элементов               */
   int items;                 /* количество элементов в списке */
} List;
```

Это снова потребует переписывания файла list.c, но программа, использующая такой список, в изменениях не нуждается.

И, наконец, подумайте о преимуществах, которые данный подход сулит процессу разработки программ. Если что-то работает не так, как следует, вполне вероятно, что проблему удастся локализовать с точностью до функции. Если удастся придумать более эффективный способ решения одной из задач, такой как добавление элемента, то придется переписать только эту одну функцию. Если требуется новая функциональная возможность, задачу можно решить путем добавления новой функции в пакет. Если окажется, что массив или двусвязный список более удобны, можно модифицировать реализацию, не изменяя программы, которые пользуются этой реализацией.

Создание очереди с помощью ADT

Как вы видели, подход к программированию на C с применением абстрактных типов данных подразумевает выполнение следующих трех шагов.

1. Описание типа, включая его операции, в абстрактной обобщенной манере.
2. Определение интерфейса в виде функций для представления нового типа.
3. Написание подробного кода для реализации интерфейса.

Этот подход был задействован при создании простого списка. Теперь воспользуемся им для построения несколько более сложного объекта — очереди.

Определение абстрактного типа данных для представления очереди

Очередь — это список, обладающий двумя особыми свойствами. Во-первых, новые элементы могут добавляться только в конец списка. В этом смысле очередь подобна простому списку. Во-вторых, элементы могут удаляться только из начала списка. Очередь можно сравнить с цепочкой людей, стоящих друг за другом в билетную кассу. Каждый новый человек становится в конец цепочки и покидает ее в самом начале после приобретения билетов. Очередь является формой данных типа *первым прибыл, первым обслужен* (first in, first out — FIFO), подобной очереди в кассу (если только никто не вклинится в очередь). Ниже дано неформальное абстрактное определение.

Имя типа: Очередь

Свойства типа: Может содержать упорядоченную последовательность элементов

Операции типа: Инициализация очереди пустым содержимым
Определение, является ли очередь пустой
Определение, является ли очередь полной
Определение количества элементов в очереди
Добавление элемента в конец очереди
Удаление и восстановление элемента в начале очереди
Опустошение очереди

Определение интерфейса

Определение интерфейса будет помещено в файл queue.h. С помощью средства typedef языка C мы создадим имена для двух типов: Item и Queue. Точная реализация соответствующих структур должна находиться в файле queue.h, но концептуально проектирование структур является частью этапа детальной реализации. А пока будем считать, что типы определены, и сосредоточим внимание на прототипах функций.

Прежде всего, следует подумать об инициализации. Она предполагает изменение типа Queue, поэтому функция должна принимать в качестве аргумента адрес переменной Queue:

```
void InitializeQueue (Queue * pq);
```

Выяснение, является очередь пустой или полной, предусматривает применение функции, которая должна возвращать истинное или ложное значение. Здесь мы будем считать, что заголовочный файл stdbool.h стандарта C99 доступен. Если это не так, можно использовать тип int или определить тип bool самостоятельно. Поскольку функция не изменяет очередь, она может принимать аргумент Queue. С другой стороны, в зависимости от реального размера объекта типа Queue, передача только адреса переменной Queue может проходить быстрее и с меньшим расходом памяти. Еще одно преимущество такого подхода заключается в том, что все функции будут принимать в качестве аргумента адрес. Для указания на то, что функции не изменяют очередь, можно (да и нужно) применять квалификатор const:

```
bool QueueIsFull(const Queue * pq);
bool QueueIsEmpty (const Queue * pq);
```

Иначе говоря, указатель pq ссылается на объект данных Queue, который не может изменяться через pq. Аналогичный прототип можно определить для функции, которая возвращает количество элементов в очереди:

```
int QueueItemCount(const Queue * pq);
```

Добавление элемента в конец очереди предусматривает идентификацию элемента и очереди. На этот раз очередь изменяется, так что использование указателя обязательно. Функция может иметь тип void либо же возвращаемое значение можно применять для указания, успешно ли выполнена операция по добавлению элемента. Давайте примем второй подход:

```
bool EnQueue(Item item, Queue * pq);
```

Наконец, удаление элемента может быть реализовано несколькими способами. Если элемент определен как структура или один из фундаментальных типов, функция может его возвращать. Аргументом функции могла бы быть переменная Queue либо указатель на нее. Таким образом, один из возможных прототипов выглядит так:

```
Item DeQueue(Queue q);
```

Однако следующий прототип является чуть более общим:

```
bool DeQueue(Item * pitem, Queue * pq);
```

Элемент, удаленный из очереди, помещается в место, на которое ссылается указатель pitem, а возвращаемое значение отражает, успешно ли выполнена операция. Единственным аргументом, который должен быть предоставлен функции опустошения очереди, является адрес очереди, что и демонстрирует приведенный далее прототип:

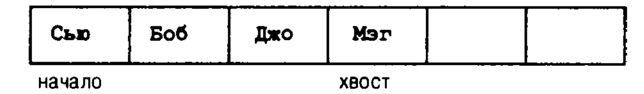
```
void EmptyTheQueue(Queue * pq);
```

Реализация представления данных интерфейса

Первый шаг предусматривает решение о том, какая форма данных С будет использоваться для очереди. Одним из вариантов является массив. Преимущества массивов связаны с простотой их применения и легкостью добавления элемента в конец заполненной части массива. Проблема возникает, когда дело доходит до удаления элемента из начала очереди. Если снова воспользоваться аналогией очереди за билетами,

удаление элемента из начала очереди заключается в копировании значения первого элемента в массиве (что просто) и последующем перемещении каждого элемента, оставшегося в массиве, на одну позицию в направлении его начала. Хотя эти действия легко программировать, они занимают много процессорного времени (рис. 17.6).

В очереди стоят четыре человека



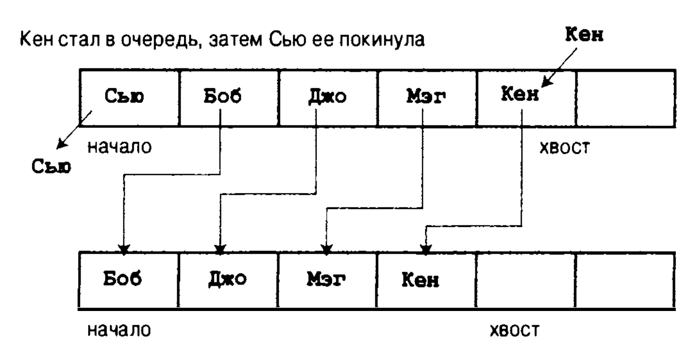


Рис. 17.6. Использование массива в качестве очереди

Второй способ решения задачи удаления в реализации с применением массива — оставить элементы в позициях, где они находятся, и затем изменить элемент, который считается начальным (рис. 17.7). Проблема этого метода в том, что место, ранее занятое удаленными элементами, расходуется впустую, что ведет к уменьшению доступного пространства в очереди.

В очереди стоят четыре человека

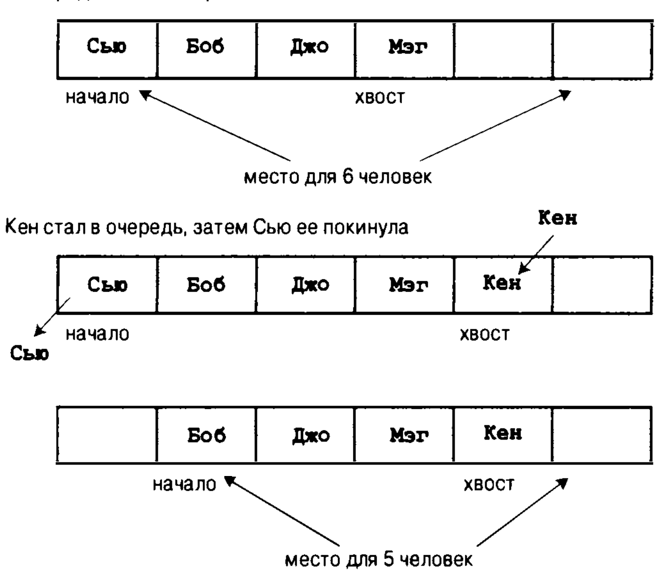


Рис. 17.7. Переопределение начального элемента

Более искусное решение проблемы теряемого пространства предполагает превращение очереди в *кольцевую*. Это означает, что конец массива должен быть соединен с его началом. То есть вообразите, что первый элемент массива следует непосредственно за последним элементом, поэтому при достижении конца массива вы начинаете добавлять элементы в начальные позиции, как если бы они были освобождены (рис. 17.8). Такой процесс можно сравнить с рисованием на бумажной ленте, склеенной в кольцо. Естественно, теперь придется выполнять дополнительные действия по обеспечению того, чтобы конец очереди не перекрывал ее начало.

Еще одно возможное решение предусматривает использование связного списка. Преимущество этого подхода состоит в том, что удаление начального элемента не требует перемещения всех остальных элементов. Взамен нужно просто переустановить указатель на начало, чтобы он указывал на новый первый элемент. Поскольку мы уже работали со связными списками, то пойдем этим путем.

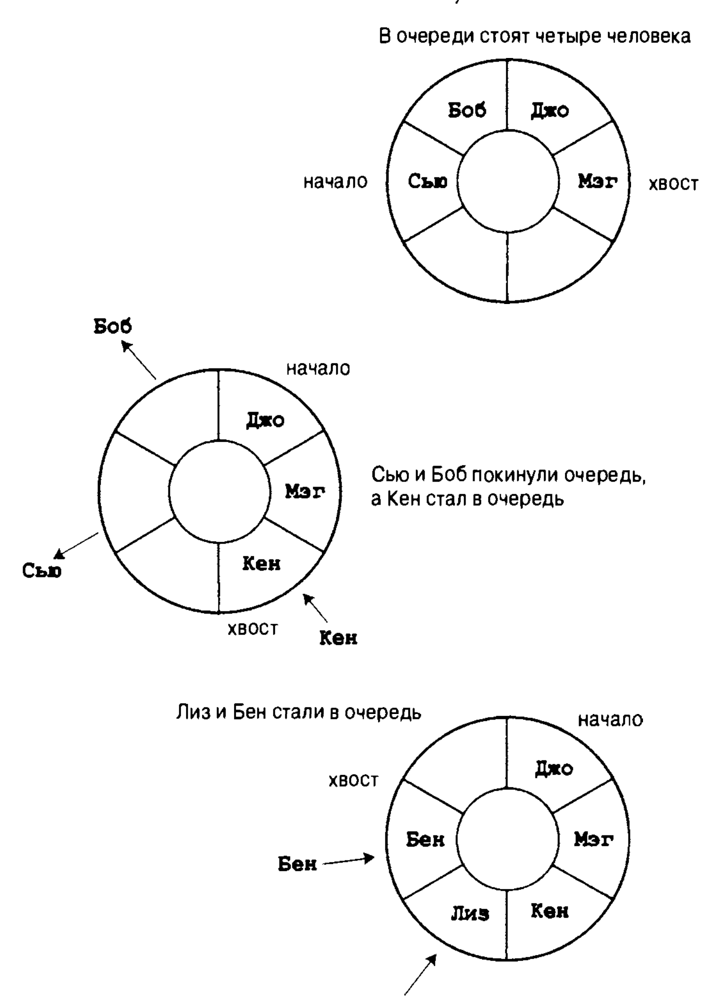


Рис. 17.8. Кольцевая очередь

Чтобы проверить свои идеи, начнем с создания очереди целых чисел:

```
typedef int Item;
```

Связный список построен из узлов, поэтому давайте определим узел:

```
typedef struct node
{
   Item item;
   struct node * next;
} Node;
```

Для очереди необходимо отслеживать начальный и конечный элементы. Это можно делать с применением указателей. Кроме того, можно использовать счетчик для отслеживания количества элементов в очереди. Таким образом, структура будет содержать два члена типа указателей и один член типа int:

```
typedef struct queue
{
   Node * front;   /* указатель на начало очереди     */
   Node * rear;    /* указатель на конец очереди      */
   int items;      /* количество элементов в очереди  */
} Queue;
```

Обратите внимание, что Queue — структура с тремя членами, поэтому ранее принятое решение об использовании в качестве аргументов указателей на очереди, а не самих очередей, экономит время и объем расходуемой памяти.

Теперь пора подумать о размере очереди. В случае связного списка размер очереди ограничен объемом доступной памяти, но часто имеет смысл применять очередь значительно меньшего размера. Например, очередь можно использовать для эмуляции самолетов, ожидающих приземления в аэропорту. Если количество ожидающих самолетов становится слишком большим, новые прибывающие самолеты могут направляться в другие аэропорты. Мы установим максимальный размер очереди равным 10. Определения и прототипы интерфейса очереди приведены в листинге 17.6. В нем отсутствует конкретное определение типа Item. Во время применения интерфейса в него будет помещено определение, соответствующее потребностям конкретной программы.

Листинг 17.6. Заголовочный файл queue.h для интерфейса очереди

```
/* queue.h -- интерфейс очереди */
#ifndef _QUEUE_H_
#define _QUEUE_H_
#include <stdbool.h>
/* ЗДЕСЬ НЕОБХОДИМО ВСТАВИТЬ ОПРЕДЕЛЕНИЕ ТИПА ITEM */
/* НАПРИМЕР, */
typedef int Item;  // для use_q.c
/* ИЛИ typedef struct item {int gumption; int charisma;} Item; */
#define MAXQUEUE 10
typedef struct node
{
   Item item;
   struct node * next;
} Node;
typedef struct queue
{
   Node * front;  /* указатель на начало очереди    */
   Node * rear;   /* указатель на конец очереди     */
   int items;     /* количество элементов в очереди */
} Queue;
```

```
/* операция:     инициализация очереди                         */
/* предусловие:  pq указывает на очередь                       */
/* постусловие:  очередь инициализирована пустым содержимым    */
void InitializeQueue(Queue * pq);

/* операция:     проверка, полна ли очередь                                   */
/* предусловие:  pq указывает на ранее инициализированную очередь             */
/* постусловие:  возвращает True, если очередь полна, и False в противном случае */
bool QueueIsFull(const Queue * pq);

/* операция:     проверка, пуста ли очередь                                   */
/* предусловие:  pq указывает на ранее инициализированную очередь             */
/* постусловие:  возвращает True, если очередь пуста, и False в противном случае */
bool QueueIsEmpty(const Queue *pq);

/* операция:     определение количества элементов в очереди     */
/* предусловие:  pq указывает на ранее инициализированную очередь */
/* постусловие:  возвращает количество элементов в очереди      */
int QueueItemCount(const Queue * pq);

/* операция:     добавление элемента в конец очереди            */
/* предусловие:  pq указывает на ранее инициализированную очередь */
/*               элемент должен быть помещен в конец очереди    */
/* постусловие:  если очередь не пуста, элемент помещается      */
/*               в конец очереди и функция возвращает True;     */
/*               в противном случае очередь остается неизменной, */
/*               а функция возвращает False                     */
bool EnQueue(Item item, Queue * pq);

/* операция:     удаление элемента из начала очереди            */
/* предусловие:  pq указывает на ранее инициализированную очередь */
/* постусловие:  если очередь не пуста, элемент в начале очереди */
/*               копируется в *pitem и удаляется из очереди,    */
/*               и функция возвращает True;                     */
/*               если операция опустошает очередь, очередь      */
/*               переустанавливается в пустое состояние.        */
/*               Если очередь пуста с самого начала, она остается */
/*               неизменной, и функция возвращает False         */
bool DeQueue(Item *pitem, Queue * pq);

/* операция:     опустошение очереди                            */
/* предусловие:  pq указывает на ранее инициализированную очередь */
/* постусловие:  очередь пуста                                  */
void EmptyTheQueue(Queue * pq);

#endif
```

Реализация функций интерфейса

Теперь можно приступить к написанию кода интерфейса. Инициализация очереди "пустым содержимым" означает установку указателей на начало и конец очереди в NULL, а счетчика (члена item) — в 0:

```
void InitializeQueue(Queue * pq)
{
   pq->front = pq->rear = NULL;
   pq->items = 0;
}
```

С помощью члена item очень легко проверить, является очередь полной или пустой, и возвратить количество элементов в очереди:

```
bool QueueIsFull(const Queue * pq)
{
   return pq->items == MAXQUEUE;
}
bool QueueIsEmpty(const Queue * pq)
{
   return pq->items == 0;
}
int QueueItemCount(const Queue * pq)
{
   return pq->items;
}
```

Добавление элемента в очередь предусматривает выполнение следующих действий.

1. Создание нового узла.
2. Копирование элемента в этот узел.
3. Установка указателя `next` этого узла в `NULL`, идентифицируя узел как последний в списке.
4. Установка указателя `next` текущего конечного узла так, чтобы он ссылался на новый узел, связывая его с очередью.
5. Установка указателя `rear` для ссылки на новый узел в целях упрощения поиска последнего узла.
6. Увеличение на 1 счетчика элементов.

Кроме того, функция должна обрабатывать два особых случая. Во-первых, если очередь пуста, указатель `front` должен быть установлен для ссылки на новый узел. Причина в том, что при наличии только одного узла этот узел является одновременно и начальным, и конечным узлом очереди. Во-вторых, если функции не удается выделить память для узла, она должна предпринять какие-то действия. Поскольку мы предполагаем использование небольших очередей, такой отказ будет возникать редко, поэтому в случае нехватки памяти функция будет просто прекращать выполнение программы. Ниже показан код функции `EnQueue()`.

```
bool EnQueue(Item item, Queue * pq)
{
   Node * pnew;
   if (QueueIsFull(pq))
      return false;
   pnew = (Node *) malloc( sizeof(Node));
   if (pnew == NULL)
   {
      fprintf(stderr, "Не удается выделить память!\n");
      exit(1);
   }
   CopyToNode(item, pnew);
   pnew->next = NULL;
   if (QueueIsEmpty(pq))
      pq->front = pnew;       /* элемент помещается в начало очереди       */
   else
      pq->rear->next = pnew;  /* связывание с концом очереди               */
   pq->rear = pnew;           /* запись местоположения конца очереди       */
   pq->items++;          /* увеличение на 1 количества элементов в очереди*/
   return true;
}
```

Функция CopyToNode() — это статическая функция, выполняющая копирование элемента в узел:

```
static void CopyToNode(Item item, Node * pn)
{
   pn->item = item;
}
```

Удаление элемента из начала очереди требует выполнения следующих действий.

1. Копирование элемента в ожидающую переменную.
2. Освобождение памяти, которая используется удаляемым узлом.
3. Переустановка указателя на начало очереди, чтобы он ссылался на следующий элемент в очереди.
4. Установка указателей на начало и конец очереди в NULL, если удален последний элемент.
5. Уменьшение на 1 счетчика элементов.

Все эти действия реализованы в показанном ниже коде:

```
bool DeQueue(Item * pitem, Queue * pq)
{
   Node * pt;

   if (QueueIsEmpty(pq))
      return false;
   CopyToItem(pq->front, pitem);
   pt = pq->front;
   pq->front = pq->front->next;
   free(pt);
   pq->items--;
   if (pq->items == 0)
      pq->rear = NULL;

   return true;
}
```

Здесь необходимо отметить пару важных фактов. Во-первых, в коде не делается явная установка указателя front в NULL, когда удаляется последний элемент. Причина в том, что указатель front уже установлен в значение указателя next удаляемого узла. Если этот узел является последним в очереди, то значение его указателя next равно NULL, поэтому указатель front получает значение NULL. Во-вторых, код использует временный указатель (pt) для отслеживания местоположения удаленного узла. Это связано с тем, что официальный указатель первого узла (pq->front) переустанавливается так, чтобы указывать на следующий узел. Поэтому без применения временного указателя программа утратила бы возможность отслеживания того, какой блок памяти освобождать.

Для опустошения очереди можно использовать функцию DeQueue(). Для этого достаточно вызывать ее в цикле до тех пор, пока очередь не станет пустой:

```
void EmptyTheQueue(Queue * pq)
{
   Item dummy;
   while (!QueueIsEmpty(pq))
      DeQueue(&dummy, pq);
}
```

НА ЗАМЕТКУ! Поддержка строгости типа ADT

После определения интерфейса ADT вы должны применить одну из его функций для поддержки типа данных. Например, обратите внимание, что функция `DeQueue()` полагается на функцию `EnQueue()` в выполнении работы по корректной установке указателей и по установке указателя `next` узла `rear` в `NULL`. Если в программе, использующей ADT, вы решите манипулировать частями очереди напрямую, это может привести к нарушению координации между функциями в пакете интерфейса.

В листинге 17.7 представлены все функции интерфейса, включая функцию `CopyToItem()`, применяемую в `EnQueue()`.

Листинг 17.7. Файл реализации queue.c

```
/* queue.c -- реализация типа Queue */
#include <stdio.h>
#include <stdlib.h>
#include "queue.h"

/* локальные функции */
static void CopyToNode(Item item, Node * pn);
static void CopyToItem(Node * pn, Item * pi);

void InitializeQueue(Queue * pq)
{
   pq->front = pq->rear = NULL;
   pq->items = 0;
}

bool QueueIsFull(const Queue * pq)
{
   return pq->items == MAXQUEUE;
}

bool QueueIsEmpty(const Queue * pq)
{
   return pq->items == 0;
}

int QueueItemCount(const Queue * pq)
{
   return pq->items;
}

bool EnQueue(Item item, Queue * pq)
{
   Node * pnew;

   if (QueueIsFull(pq))
      return false;
   pnew = (Node *) malloc( sizeof(Node));
   if (pnew == NULL)
   {
      fprintf(stderr, "Не удается выделить память!\n");
      exit(1);
   }
   CopyToNode(item, pnew);
   pnew->next = NULL;
   if (QueueIsEmpty(pq))
      pq->front = pnew;         /* элемент помещается в начало очереди         */
   else
      pq->rear->next = pnew;    /* связывание с концом очереди                 */
```

```
   pq->rear = pnew;            /* запись местоположения конца очереди          */
   pq->items++;                /* увеличение на 1 количества элементов в очереди */

   return true;
}
bool DeQueue(Item * pitem, Queue * pq)
{
   Node * pt;
   if (QueueIsEmpty(pq))
      return false;
   CopyToItem(pq->front, pitem);
   pt = pq->front;
   pq->front = pq->front->next;
   free(pt);
   pq->items--;
   if (pq->items == 0)
      pq->rear = NULL;

   return true;
}
/* опустошение очереди */
void EmptyTheQueue(Queue * pq)
{
   Item dummy;
   while (!QueueIsEmpty(pq))
      DeQueue(&dummy, pq);
}

/* локальные функции */

static void CopyToNode(Item item, Node * pn)
{
   pn->item = item;
}
static void CopyToItem(Node * pn, Item * pi)
{
   *pi = pn->item;
}
```

Тестирование очереди

Прежде чем включать новую структуру, такую как пакет очереди, в важную программу, эту структуру необходимо протестировать. Один из подходов к тестированию предусматривает создание короткой программы, иногда называемой *драйвером*, единственное назначение которой состоит в тестировании пакета. Например, в коде, приведенном в листинге 17.8, очередь используется для добавления и удаления целых чисел. Прежде чем компилировать программу, убедитесь в наличии следующей строки в файле queue.h:

```
typedef int item;
```

Кроме того, не забудьте о необходимости выполнения компоновки с queue.c и use_q.c.

Листинг 17.8. Программа use_q.c

```
/* use_q.c -- тестирование интерфейса Queue с помощью драйвера */
/* компилировать вместе с queue.c                         */
#include <stdio.h>
#include "queue.h"  /* определение Queue, Item            */
```

```
int main(void)
{
   Queue line;
   Item temp;
   char ch;

   InitializeQueue(&line);
   puts("Тестирование интерфейса Queue. Введите a, чтобы добавить значение,");
   puts("введите d, чтобы удалить значение, или введите q для выхода из программы.");
   while ((ch = getchar()) != 'q')
   {
     if (ch != 'a' && ch != 'd')   /* игнорировать другие вводимые данные */
         continue;
     if ( ch == 'a')
     {
         printf("Целое число для добавления: ");
         scanf("%d", &temp);
         if (!QueueIsFull(&line))
         {
            printf("Помещение %d в очередь\n", temp);
            EnQueue(temp,&line);
         }
        else
           puts("Очередь полна!");
     }
     else
     {
         if (QueueIsEmpty(&line))
            puts("Элементы для удаления отсутствуют!");
         else
         {
             DeQueue(&temp,&line);
             printf("Удаление %d из очереди\n", temp);
         }
     }
     printf("%d элемент(ов) в очереди\n", QueueItemCount(&line));
     puts("Введите a, чтобы добавить, d, чтобы удалить, или q для выхода из программы:");
   }
   EmptyTheQueue(&line);
   puts("Программа завершена.");

   return 0;
}
```

Ниже показаны результаты пробного запуска. Вы должны также протестировать корректность работы реализации в случае, когда очередь полна.

```
Тестирование интерфейса Queue. Введите a, чтобы добавить значение,
введите d, чтобы удалить значение, или введите q для выхода из программы.
a
Целое число для добавления: 40
Помещение 40 в очередь
1 элемент(ов) в очереди
Введите a, чтобы добавить, d, чтобы удалить, или q для выхода из программы:
a
Целое число для добавления: 20
Помещение 20 в очередь
2 элемент(ов) в очереди
```

```
Введите a, чтобы добавить, d, чтобы удалить, или q для выхода из программы:
a
Целое число для добавления: 55
Помещение 55 в очередь
3 элемент(ов) в очереди
Введите a, чтобы добавить, d, чтобы удалить, или q для выхода из программы:
d
Удаление 40 из очереди
2 элемент(ов) в очереди
Введите a, чтобы добавить, d, чтобы удалить, или q для выхода из программы:
d
Удаление 20 из очереди
1 элемент(ов) в очереди
Введите a, чтобы добавить, d, чтобы удалить, или q для выхода из программы:
d
Удаление 55 из очереди
0 элемент(ов) в очереди
Введите a, чтобы добавить, d, чтобы удалить, или q для выхода из программы:
d
Элементы для удаления отсутствуют!
0 элемент(ов) в очереди
Введите a, чтобы добавить, d, чтобы удалить, или q для выхода из программы:
q
Программа завершена.
```

Моделирование реальной очереди

Итак, тип очереди работает! Давайте теперь с его помощью решим какую-то более интересную задачу. Очереди встречаются во многих реальных ситуациях. Это могут быть, к примеру, очереди клиентов в банках и универсамах, очереди самолетов в аэропортах и очереди задач в многозадачных компьютерных системах. Пакет очереди можно применять для моделирования ситуаций подобного рода.

Предположим, что некий Зигмунд Ландер установил консультационный киоск в торговом центре. Клиенты могут заплатить за одну, две или три минуты консультаций. Для обеспечения свободного прохода действующие в торговом центре правила ограничивают количество клиентов в очереди до 10 (что легко определяет максимальный размер очереди в программе). Представим, что люди подходят к киоску случайным образом, а время, которое они тратят на получение консультации, произвольно распределяется между тремя возможными вариантами (одна, две или три минуты). Сколько в среднем клиентов придется обслужить Зигмунду в течение часа? Сколько в среднем каждому клиенту придется дожидаться своей очереди? Какой будет средняя длина очереди? Моделирование может дать ответы на вопросы такого рода.

Прежде всего, давайте решим, что именно помещать в очередь. Каждого клиента можно описывать в терминах времени, когда он становится в очередь, и количества минут, которые он собирается потратить на консультацию. Это предполагает следующее определение элемента `Item`:

```
typedef struct item
{
   long arrive;      /* время присоединения клиента к очереди  */
   int processtime;  /* желаемое количество минут консультации */
} Item;
```

Для преобразования пакета очереди, чтобы он обрабатывал эту структуру, а не тип `int`, использованный в последнем примере, достаточно заменить предыдущее опреде-

ление typedef типа Item приведенным выше. После этого вам не придется беспокоиться о деталях функционирования очереди. Вместо этого вы сможете сосредоточить все внимание на реальной задаче – моделировании очереди к киоску Зигмунда.

Рассмотрим один из возможных подходов. Пусть отсчет времени осуществляется одноминутными интервалами. Тогда каждую минуту необходимо проверять, не появился ли новый клиент. Если клиент подошел, и очередь не переполнена, клиента необходимо добавить в очередь. Это предусматривает запись в структуру Item времени прибытия клиента и длительности консультации, которую клиент желает оплатить, с последующим добавлением элемента в очередь. Однако если очередь полна, клиента нужно отправить. В целях учета мы будем отслеживать общее число клиентов и общее количество "отказов" (людей, которые не могут стать в очередь, поскольку она переполнена).

Далее потребуется обработать начало очереди. То есть, если очередь не пуста и Зигмунд не занят обслуживанием предыдущего клиента, необходимо удалить элемент из начала очереди. Вспомните, что элемент содержит показание времени присоединения клиента к очереди. Сравнивая это показание с текущим временем, мы получаем время нахождения клиента в очереди (в минутах). Элемент содержит также количество минут, в течение которых клиент желает получить консультацию; это значение определяет интервал, на протяжении которого Зигмунд будет занят обслуживанием нового клиента. Для отслеживания времени ожидания мы применяем переменную. Если Зигмунд занят, из очереди никто не удаляется, но значение переменной для отслеживания времени ожидания должно декрементироваться.

Основной код может выглядеть похожим на показанный далее, при этом каждый цикл соответствует одной минуте активности:

```
for (cycle = 0; cycle < cyclelimit; cycle++)
{
    if (newcustomer(min_per_cust))
    {
        if (QueueIsFull(&line))
            turnaways++;
        else
        {
            customers++;
            temp = customertime(cycle);
            EnQueue(temp, &line);
        }
    }
    if (wait_time <= 0 && !QueueIsEmpty(&line))
    {
        DeQueue (&temp, &line);
        wait_time = temp.processtime;
        line_wait += cycle - temp.arrive;
        served++;
    }
    if (wait_time > 0)
        wait_time--;
    sum_line += QueueItemCount(&line);
}
```

Обратите внимание, что разрешение времени является относительно грубым (одна минута), так что максимальное количество клиентов в час составляет всего 60.

Ниже представлены краткие описания некоторых переменных и функций.

- min_per_cust — среднее количество минут между прибытиями клиентов.
- newcustomer() использует функцию rand() языка С для определения, появляется ли клиент в течение этой конкретной минуты.
- turnaways — количество прибывших клиентов, которым было отказано в обслуживании.
- customers — количество прибывающих клиентов, которые становятся в очередь.
- temp — переменная типа Item, описывающая нового клиента.
- customertime() устанавливает члены arrive и processtime структуры temp.
- wait_time — количество минут, остающееся до того момента, когда Зигмунд завершит консультирование текущего клиента.
- line_wait — накапливаемое значение времени, потраченное в очереди всеми клиентами на текущий момент.
- served — количество действительно обслуженных клиентов.
- sum_line — накапливаемое значение длины очереди на текущий момент.

Только подумайте, насколько более запутанным и непонятным выглядел бы код, если бы он оказался усыпанным вызовами функций malloc() и free() и указателями на узлы. Наличие пакета очереди позволяет сосредоточиться на задаче моделирования, не отвлекаясь на детали программирования.

Полный код для моделирования консультационного киоска в торговом центре представлен в листинге 17.9. В соответствие с методом, предложенным в главе 12, для генерации случайных значений применяются стандартные функции rand(), srand() и time(). Чтобы можно было использовать программу, обновите определение типа Item в файле queue.h следующим образом:

```
typedef struct item
{
   long arrive;      // время присоединения клиента к очереди
   int processtime;  // желаемое количество минут консультации
} Item;
```

Не забудьте также выполнить компоновку mall.c с queue.c.

Листинг 17.9. Программа mall.c

```
// mall.c -- использует интерфейс Queue
// компилировать вместе с queue.c
#include <stdio.h>
#include <stdlib.h>          // для rand() и srand()
#include <time.h>            // для time()
#include "queue.h"           // измените определение типа Item
#define MIN_PER_HR 60.0

bool newcustomer(double x);    // имеется новый клиент?
Item customertime(long when);  // установка параметров клиента

int main(void)
{
   Queue line;
   Item temp;                  // данные о новом клиенте
   int hours;                  // количество часов моделирования
   int perhour;                // среднее количество прибывающих клиентов в час
```

```
    long cycle, cyclelimit;// счетчик и граничное значение цикла
    long turnaways = 0;     // количество отказов из-за переполненной очереди
    long customers = 0;     // количество клиентов присоединившихся к очереди
    long served = 0;        // количество клиентов, обслуженных за время моделирования
    long sum_line = 0;      // накопительное значение длины очереди
    int wait_time = 0;      // время до освобождения Зигмунда
    double min_per_cust;    // среднее время между прибытиями клиентов
    long line_wait = 0;     // накопительное значение времени в очереди
    InitializeQueue(&line);
    srand((unsigned int) time(0)); // случайная инициализация rand()
    puts("Учебный пример: консультационный киоск Зигмунда Ландера");
    puts("Введите длительность моделирования в часах:");
    scanf("%d", &hours);
    cyclelimit = MIN_PER_HR * hours;
    puts("Введите среднее количество клиентов, прибывающих за час:");
    scanf("%d", &perhour);
    min_per_cust = MIN_PER_HR / perhour;
    for (cycle = 0; cycle < cyclelimit; cycle++)
    {
        if (newcustomer(min_per_cust))
        {
            if (QueueIsFull(&line))
                turnaways++;
            else
            {
                customers++;
                temp = customertime(cycle);
                EnQueue(temp, &line);
            }
        }
        if (wait_time <= 0 && !QueueIsEmpty(&line))
        {
            DeQueue (&temp, &line);
            wait_time = temp.processtime;
            line_wait += cycle - temp.arrive;
            served++;
        }
        if (wait_time > 0)
            wait_time--;
        sum_line += QueueItemCount(&line);
    }
    if (customers > 0)
    {
        printf("    принятых клиентов: %ld\n", customers);
        printf("  обслуженных клиентов: %ld\n", served);
        printf("              отказов: %ld\n", turnaways);
        printf(" средняя длина очереди: %.2f\n",
              (double) sum_line / cyclelimit);
        printf("среднее время ожидания: %.2f мин\n",
              (double) line_wait / served);
    }
    else
        puts("Клиенты отсутствуют!");
    EmptyTheQueue(&line);
    puts("Программа завершена.");
    return 0;
}
```

```
// x - среднее время между прибытием клиентов в минутах
// возвращает true, если клиент появляется в течение данной минуты
bool newcustomer(double x)
{
    if (rand() * x / RAND_MAX < 1)
        return true;
    else
        return false;
}

// when - время прибытия клиента
// функция возвращает структуру Item со временем прибытия,
// установленным в when, и временем обслуживания,
// установленным в случайное значение из диапазона от 1 до 3
Item customertime(long when)
{
    Item cust;

    cust.processtime = rand() % 3 + 1;
    cust.arrive = when;

    return cust;
}
```

Программа позволяет указывать количество часов моделирования и среднее число клиентов, обращающихся за консультацией в течение часа. Выбор большого количества часов моделирования обеспечит получение довольно точных средних значений, тогда как малое количество часов дает своего рода случайную вариацию, которая может иметь место от часа к часу. Эти моменты демонстрируют показанные ниже результаты пробных запусков. Обратите внимание, что средние значения длины очереди и времени ожидания для 80 часов и для 800 часов почти совпадают, но результаты двух одночасовых выборок существенно отличаются как друг от друга, так и от средних значений для более длительных периодов. Это обусловлено тем, что меньшие статистические выборки характеризуются большими относительными вариациями.

```
Учебный пример: консультационный киоск Зигмунда Ландера
Введите длительность моделирования в часах:
80
Введите среднее количество клиентов, прибывающих за час:
20
     принятых клиентов: 1633
  обслуженных клиентов: 1633
               отказов: 0
 средняя длина очереди: 0.46
среднее время ожидания: 1.35 мин

Учебный пример: консультационный киоск Зигмунда Ландера
Введите длительность моделирования в часах:
800
Введите среднее количество клиентов, прибывающих за час:
20
     принятых клиентов: 16020
  обслуженных клиентов: 16019
               отказов: 0
 средняя длина очереди: 0.44
среднее время ожидания: 1.32 мин

Учебный пример: консультационный киоск Зигмунда Ландера
```

```
Введите длительность моделирования в часах:
1
Введите среднее количество клиентов, прибывающих за час:
20
      принятых клиентов: 20
  обслуженных клиентов: 20
                отказов: 0
 средняя длина очереди: 0.23
среднее время ожидания: 0.70 мин
```

```
Учебный пример: консультационный киоск Зигмунда Ландера
Введите длительность моделирования в часах:
1
Введите среднее количество клиентов, прибывающих за час:
20
      принятых клиентов: 22
  обслуженных клиентов: 22
                отказов: 0
 средняя длина очереди: 0.75
среднее время ожидания: 2.05 мин
```

Еще один способ применения этой программы предусматривает сохранение длительности моделирования неизменной, но указание разных средних значений числа клиентов, прибывающих в течение часа. Ниже приведены результаты двух пробных запусков программы для исследования такой вариации.

```
Учебный пример: консультационный киоск Зигмунда Ландера
Введите длительность моделирования в часах:
80
Введите среднее количество клиентов, прибывающих за час:
25
      принятых клиентов: 1960
  обслуженных клиентов: 1959
                отказов: 3
 средняя длина очереди: 1.43
среднее время ожидания: 3.50 мин
```

```
Учебный пример: консультационный киоск Зигмунда Ландера
Введите длительность моделирования в часах:
80
Введите среднее количество клиентов, прибывающих за час:
30
      принятых клиентов: 2376
  обслуженных клиентов: 2373
                отказов: 94
 средняя длина очереди: 5.85
среднее время ожидания: 11.83 мин
```

Обратите внимание на резкое возрастание среднего времени ожидания с увеличением частоты прибытия клиентов. Среднее время ожидания при 20 клиентах в час (80-часовое моделирование) составило 1,35 минуты. Это значение возрастает до 3,5 минуты при 25 клиентах в час и до 11,83 минуты при 30 клиентах в час. Кроме того, количество отказов возрастает от 0 до 3 и до 94 соответственно. Зигмунд мог бы воспользоваться подобным анализом для принятия решения о необходимости открытия второго киоска.

Сравнение связного списка и массива

Многие задачи программирования, такие как создание списка или очереди, могут решаться с помощью связного списка, под которым мы понимаем связанную последовательность динамически выделяемых структур, или посредством массива. Каждая форма обладает преимуществами и недостатками, поэтому выбор между ними зависит от конкретных требований задачи. Основные характеристики связных списков и массивов приведены в табл. 17.1.

Таблица 17.1. Сравнение массивов и связных списков

Форма данных	Достоинства	Недостатки
Массив	Напрямую поддерживается в С. Предоставляет произвольный доступ к элементам данных	Размер определяется во время компиляции. Вставка и удаление элементов требуют значительных затрат времени
Связный список	Размер определяется во время выполнения. Вставка и удаление выполняются быстро	Произвольный доступ к элементам невозможен. Пользователь должен обеспечить программную поддержку

Давайте более пристально взглянем на процесс вставки и удаления элементов. Для вставки элемента в массив необходимо переместить элементы, чтобы освободить место под новый элемент (рис. 17.9). Чем ближе к началу массива должен быть помещен новый элемент, тем больше элементов потребуется переместить. В то же время для вставки узла в связный список достаточно присвоить значения двум указателям (рис. 17.10). Аналогично, удаление элемента из массива требует полного изменения расположения элементов, а для удаления узла из связного списка достаточно переустановки указателя и освобождения памяти, которую занимал удаленный узел.

Теперь посмотрим, как получить доступ к элементам списка. В массиве для непосредственного обращения к любому элементу можно применять индекс массива. Это называется *произвольным доступом*. В связном списке необходимо начинать с начала списка и затем переходить от узла к узлу до тех пор, пока не будет достигнут желаемый узел; это называется *последовательным доступом*.

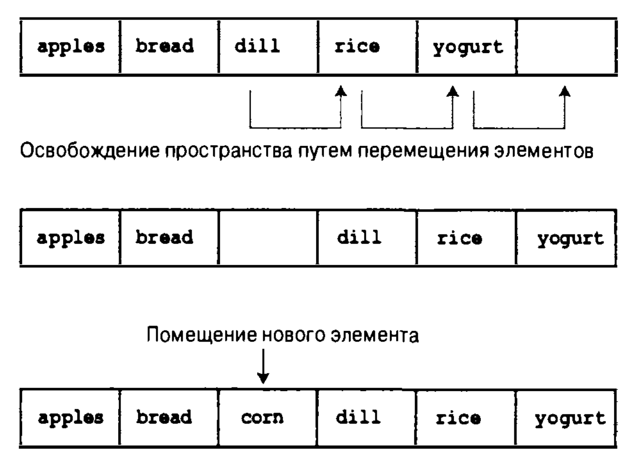


Рис. 17.9. Вставка элемента в массив

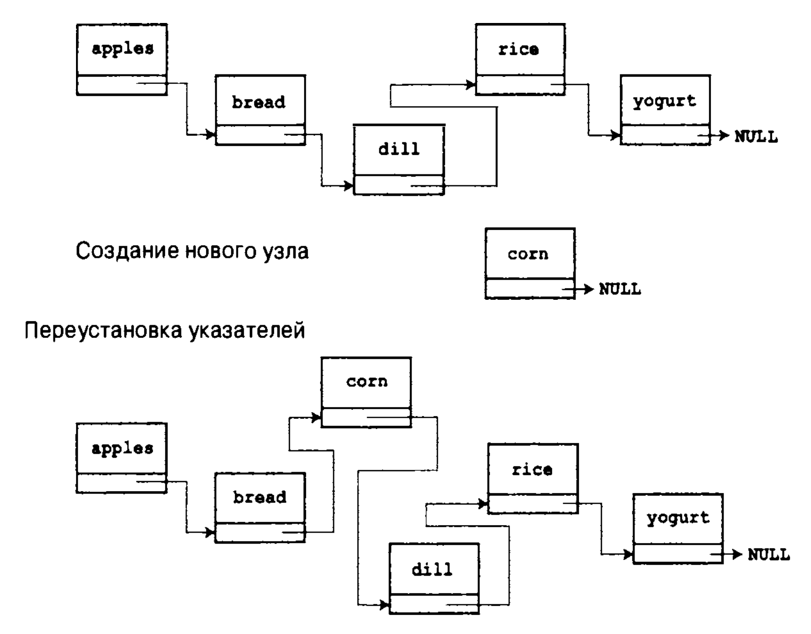


Рис. 17.10. *Вставка элемента в связный список*

Последовательный доступ может быть также реализован и в массиве. Для упорядоченного перемещения по массиву достаточно инкрементировать его индекс. В одних ситуациях последовательного доступа вполне достаточно. Например, если требуется отобразить каждый элемент в списке, то последовательный доступ прекрасно подойдет. В других ситуациях, как будет показано далее, наличие произвольного доступа дает огромное преимущество.

Предположим, что в списке необходимо найти конкретный элемент. Один из возможных алгоритмов предусматривает старт поиска с начала списка и последовательный просмотр его элементов; это называется *последовательным поиском.* Если элементы не упорядочены каким-либо образом, то последовательный поиск — это практически все, что можно предпринять. Если искомый элемент в списке отсутствует, придется просмотреть все элементы, прежде чем можно будет утверждать об этом. (Здесь может помочь параллельное программирование, т.к. разные процессоры могут выполнять поиск в разных частях списка одновременно.)

Последовательный поиск можно улучшить, предварительно отсортировав список. Это позволяет прервать поиск, если искомый элемент не найден по достижении элемента, который должен был бы следовать за искомым. Например, предположим, что мы выполняем поиск элемента *Susan* в списке, упорядоченном по алфавиту, и через некоторое время наталкиваемся на элемент *Sylvia,* так и не найдя элемента *Susan.* В этом месте поиск можно прервать, поскольку элемент *Susan,* если бы присутствовал в списке, то он предшествовал бы элементу *Sylvia.* В среднем этот метод может вдвое сократить время поиска элементов, отсутствующих в списке.

В случае упорядоченного списка для поиска можно использовать намного более эффективный метод *двоичного поиска.* Вот как он работает. Для начала назовем искомый элемент списка *целевым* и предположим, что список упорядочен по алфавиту. Затем выберем элемент, расположенный посередине списка, и сравним его с целевым элементом. Если эти два элемента равны, поиск завершен. Если элемент списка в алфавитном порядке предшествует целевому элементу, то целевой элемент, если он

присутствует в списке, должен находиться во второй половине. Если элемент списка следует за целевым, то целевой элемент должен располагаться в первой половине. В любом случае правила поиска уменьшают количество просматриваемых элементов вдвое. Затем этот метод применяется снова. То есть мы выбираем элемент, расположенный посередине остающейся половины списка. Как и ранее, метод либо находит элемент, либо вдвое уменьшает размер просматриваемого списка. Эти действия продолжаются до тех пор, пока элемент не будет найден или пока не будет исключен весь список (рис. 17.11). Описанный метод весьма эффективен. Для примера предположим, что список содержит 127 элементов. При использовании последовательного поиска обнаружение элемента либо установление его отсутствия в списке требовало бы в среднем 64 операции сравнения. В то же время двоичный поиск требовал бы выполнения не более 7 сравнений. Первая операция сравнения уменьшает количество возможных совпадений до 63, вторая – до 31 и т.д., пока шестое сравнение не уменьшит число возможных элементов до 1. После этого седьмая операция сравнения определяет, является ли остающийся элемент целевым.

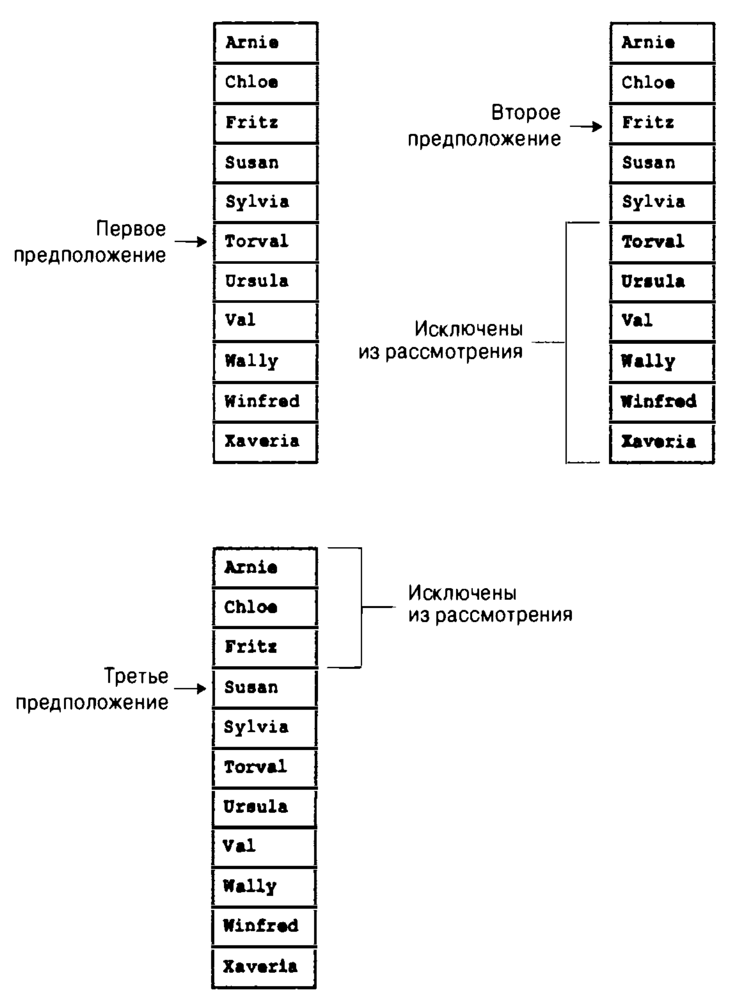


Рис. 17.11. Двоичный поиск элемента Susan

В общем случае n сравнений позволяют обработать массив, содержащий 2^n-1 элементов, поэтому преимущество применения двоичного поиска по сравнению с последовательным поиском становится все более явным по мере увеличения длины списка.

Реализовать двоичный поиск в массиве довольно просто, т.к. для определения средней точки в любом списке или его части можно использовать индекс массива. Для этого нужно сложить индексы начального и конечного элементов части списка и разделить результат на 2.

Например, в списке, состоящем из 100 элементов, первый индекс равен 0, а последний — 99, и начальным предположением является (0 + 99)/2, или 49 (целочисленное деление). Если элемент с индексом 49 располагается слишком далеко по алфавиту, искомый элемент должен находиться в диапазоне 0–48, поэтому вторым предположением становится индекс (0 + 48)/2, или 24. Если 24-й элемент в алфавитном порядке расположен слишком близко, то следующим предположением будет индекс (25 + 48)/2, или 36. Именно здесь в игру вступает возможность произвольного доступа к элементам массива. Она позволяет переходить от одного элемента к другому, не посещая все расположенные между ними элементы. Связные списки, которые поддерживают только последовательный доступ, не предоставляют средства для перехода к точке в середине списка, поэтому прием двоичного поиска нельзя применять к связным спискам.

Как видите, выбор типа данных зависит от решаемой задачи. Если ситуация требует использования списка, размер которого постоянно изменяется за счет частых вставок и удалений элементов, но поиск в котором производится не особенно часто, то лучше выбрать связный список. В тех же ситуациях, когда необходим стабильный список с редкими вставками и удалениями, но частым поиском, лучше применять массив.

А что, если требуется форма данных, поддерживающая как частые вставки и удаления, так и частый поиск? Ни связный список, ни массив далеко не идеальны для таких целей. Наиболее подходящей может оказаться другая форма данных — двоичное дерево поиска.

Двоичные деревья поиска

Двоичное дерево поиска — это связная структура, которая включает в себя поддержку стратегии двоичного поиска. Каждый узел дерева содержит элемент и два указателя на другие узлы, называемые *дочерними узлами*. Связь между узлами в двоичном дереве поиска показана на рис. 17.12. Основная идея этой структуры состоит в том, что каждый узел имеет два дочерних узла — левый и правый. Порядок элементов определяется тем, что элемент в левом узле предшествует элементу в родительском узле, а элемент в правом узле следует за элементом родительского узла. Это отношение сохраняется для всех узлов среди дочерних узлов. Более того, все элементы, чья родословная может быть прослежена до левого узла родительского узла, содержат элементы, которые предшествуют родительскому элементу, а все элементы, являющиеся потомками правого узла, содержат элементы, следующие за родительским элементом. Слова в дереве на рис. 17.12 хранятся именно таким образом. Верхняя часть дерева, в отличие от ботаники, называется *корнем*. Дерево представляет собой *иерархическую* организацию данных, т.е. данные организованы по рангам, или уровням, причем в общем случае каждому рангу соответствуют ранги, расположенные над и под ним. Если двоичное дерево поиска полностью заполнено, каждый уровень содержит вдвое больше узлов, чем уровень, расположенный над ним.

Каждый узел в двоичном дереве поиска сам является корнем узлов, исходящих из него, что превращает этот узел и его потомков в *поддерево*.

Например, на рис. 17.12 узлы, содержащие слова *fate*, *carpet* и *llama*, образуют левое поддерево всего дерева, а слово *voyage* является правым поддеревом поддерева *style-plenum-voyage*.

Предположим, что в таком дереве необходимо найти элемент – назовем его *целевым*. Если элемент предшествует корневому элементу, поиск понадобится выполнять только в левой половине дерева. Если же целевой элемент следует за корневым элементом, то поиск должен выполняться только в правом поддереве корневого узла. Таким образом, одно сравнение исключает из поиска половину дерева.

Предположим, что поиск осуществляется в левой половине. Это означает сравнение целевого элемента с элементом в левом дочернем узле. Если целевой элемент предшествует элементу левого дочернего узла, то поиск необходимо выполнять только в левой половине дочерних узлов и т.д. Как и при двоичном поиске, каждое сравнение уменьшает количество потенциальных сопоставлений в два раза.

Давайте применим этот метод, чтобы выяснить, присутствует ли слово *puppy* в дереве, показанном на рис. 17.12. Сравнивая слово *puppy* с *melon* (элементом корневого узла) мы видим, что слово *puppy*, если оно присутствует, должно располагаться в правой половине дерева. Поэтому мы переходим к правому дочернему узлу корневого узла и сравниваем *puppy* со словом *style*. В данном случае целевой элемент предшествует элементу узла, поэтому следует двигаться по связи к левому узлу. Здесь находится слово *plenum*, которое предшествует слову *puppy*. Теперь необходимо следовать правой ветвью для этого узла, но она пуста. Таким образом, три операции сравнения позволили установить, что слово *puppy* в дереве отсутствует.

Таким образом, двоичное дерево поиска сочетает преимущества связной структуры с эффективностью двоичного поиска. С точки зрения программирования реализация дерева является более трудоемким процессом, чем создание связного списка. Далее мы построим двоичное дерево для финального проекта ADT.

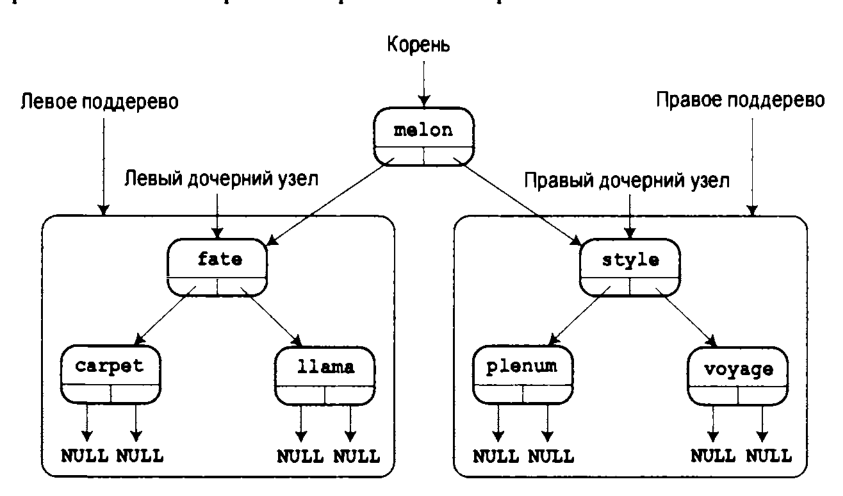


Рис. 17.12. Хранение слов в двоичном дереве поиска

Создание абстрактного типа данных для двоичного дерева

Как обычно, мы начнем с общего определения двоичного дерева. Это конкретное определение предполагает, что дерево не содержит дублированных элементов. Многие его операции совпадают с операциями со списками. Различие состоит в иерархической организации данных.

Неформальное определение этого типа ADT выглядит следующим образом.

Имя типа: Двоичное дерево поиска

Свойства типа: Двоичное дерево является либо пустым набором узлов (пустое дерево), либо набором узлов, один из которых обозначает корень.

Каждый узел имеет в точности два исходящих из него дерева, называемые *левым поддеревом* и *правым поддеревом*.

Каждое поддерево само является двоичным деревом, включая возможность быть пустым деревом.

Двоичное дерево поиска — это упорядоченное двоичное дерево, где каждый узел содержит элемент, внутри которого все элементы в левом поддереве предшествуют корневому элементу, а корневой элемент предшествует всем элементам в правом поддереве.

Операции типа: Инициализация дерева пустым значением

Определение, является ли дерево пустым

Определение, является ли дерево полным

Определение количества элементов в дереве

Добавление элемента в дерево

Удаление элемента из дерева

Поиск элемента в дереве

Посещение каждого элемента в дереве

Опустошение дерева

Интерфейс двоичного дерева поиска

В принципе, двоичное дерево поиска можно реализовать разнообразными способами. Его можно реализовать даже в виде массива, соответствующим образом манипулируя индексами массива. Но наиболее прямолинейный способ реализации двоичного дерева поиска предполагает использование динамически выделяемых узлов, связанных между собой посредством указателей, так что давайте начнем с определений вроде показанных ниже:

```
typedef SOMETHING Item;

typedef struct trnode
{
   Item item;
   struct trnode * left;
   struct trnode * right;
} Trn;

typedef struct tree
{
   Trnode * root;
   int size;
} Tree;
```

Каждый узел содержит элемент, указатель на левый дочерний узел и указатель на правый дочерний узел. Структуру `Tree` можно было бы определить как тип указателя на `Trnode`, поскольку для доступа ко всему дереву достаточно знать только местоположение корневого узла. Однако применение структуры с членом `size` упрощает отслеживание размера дерева.

Мы разработаем программу для ведения реестра домашних животных в клубе Nelville Pet Club, причем каждый элемент будет состоять из клички животного и его вида. Учитывая это, мы можем определить интерфейс, показанный в листинге 17.10. Мы ограничили размер дерева до 10. Небольшой размер облегчает тестирование программы при заполнении дерева. При необходимости значение MAXITEMS всегда можно увеличить.

Листинг 17.10. Заголовочный файл tree.h для интерфейса двоичного дерева поиска

```
/* tree.h -- двоичное дерево поиска                          */
/*        дублированные элементы в этом дереве не разрешены */
#ifndef _TREE_H_
#define _TREE_H_
#include <stdbool.h>

/* переопределение типа Item подходящим образом */
typedef struct item
{
   char petname[20];
   char petkind[20];
} Item;

#define MAXITEMS 10

typedef struct trnode
{
   Item item;
   struct trnode * left;      /* указатель на левую ветвь    */
   struct trnode * right;     /* указатель на правую ветвь   */
} Trnode;

typedef struct tree
{
   Trnode * root;          /* указатель на корень дерева      */
   int size;               /* количество элементов в дереве   */
} Tree;

/* прототипы функций */

/* операция:      инициализация дерева пустым содержимым      */
/* предусловия:   ptree указывает на дерево                   */
/* постусловия:   дерево установлено в пустое состояние       */
void InitializeTree(Tree * ptree);

/* операция:      определение, является ли дерево пустым      */
/* предусловия:   ptree указывает на дерево                   */
/* постусловия:   функция возвращает true, если дерево        */
/*                пустое, и false — в противном случае        */
bool TreeIsEmpty(const Tree * ptree);

/* операция:      определение, является ли дерево полным      */
/* предусловия:   ptree указывает на дерево                   */
/* постусловия:   функция возвращает true, если дерево        */
/*                полное, и false — в противном случае        */
bool TreeIsFull(const Tree * ptree);

/* операция:      определение количества элементов в дереве   */
/* предусловия:   ptree указывает на дерево                   */
/* постусловия:   функция возвращает количество элементов в дереве */
int TreeItemCount(const Tree * ptree);

/* операция:      добавление элемента к дереву                */
/* предусловия:   pi — адрес добавляемого элемента            */
/*                ptree указывает на инициализированное дерево */
```

```
/* постусловия:  если возможно, функция добавляет элемент               */
/*               к дереву и возвращает true;                            */
/*               в противном случае она возвращает false                */
bool AddItem(const Item * pi, Tree * ptree);

/* операция:     поиск элемента в дереве                                */
/* предусловия:  pi указывает на элемент                                */
/*               ptree указывает на инициализированное дерево           */
/* постусловия:  функция возвращает true, если элемент присутствует     */
/*               в дереве, и false — в противном случае                 */
bool InTree(const Item * pi, const Tree * ptree);

/* операция:     удаление элемента из дерева                            */
/* предусловия:  pi — адрес удаляемого элемента                         */
/*               ptree указывает на инициализированное дерево           */
/* постусловия:  если возможно, функция удаляет элемент из дерева       */
/*               и возвращает true; в противном случае функция          */
/*               возвращает false                                       */
bool DeleteItem(const Item * pi, Tree * ptree);

/* операция:     применение указанной функции к каждому элементу в дереве */
/* предусловия:  ptree указывает на дерево                              */
/*               pfun указывает на функцию, которая принимает           */
/*               аргумент Item и не имеет возвращаемого значения        */
/* постусловия:  функция, указанная с помощью pfun, выполняется один раз */
/*               для каждого элемента в дереве                          */
void Traverse (const Tree * ptree, void (* pfun)(Item item));

/* операция:     удаление всех элементов из дерева                      */
/* предусловия:  ptree указывает на инициализированное дерево           */
/* постусловия:  дерево является пустым                                 */
void DeleteAll(Tree * ptree);

#endif
```

Реализация двоичного дерева

Теперь приступим к реализации множества функций, описанных в файле `tree.h`. Функции `InitializeTree()`, `EmptyTree()`, `FullTree()` и `TreeItems()` достаточно просты и работают подобно своим аналогам в абстрактных типах данных списка и очереди, поэтому мы уделим основное внимание остальным функциям.

Добавление элемента

При добавлении элемента в дерево сначала потребуется проверить, есть ли место для нового узла. Поскольку двоичное дерево поиска определено так, что не может содержать дублированных элементов, далее необходимо выяснить, не присутствует ли данный элемент в дереве. Если новый элемент удовлетворяет этим двум начальным условиям, нужно создать новый узел, скопировать в него элемент и установить левый и правый указатели узла в `NULL`. Это говорит об отсутствии дочерних узлов у дочернего узла. Затем следует обновить элемент `size` структуры `Tree` с целью отражения добавления нового элемента. Далее понадобится выяснить, в какую позицию дерева должен быть помещен новый узел. Если дерево пустое, корневой указатель необходимо установить так, чтобы он ссылался на новый узел. В противном случае потребуется просмотреть дерево, чтобы найти в нем место для добавления узла. Функция `AddItem()` выполняет эти действия, передавая часть работы функциям, которые пока еще не определены: `SeekItem()`, `MakeNode()` и `AddNode()`.

```
bool AddItem(const Item * pi, Tree * ptree)
{
   Trnode * new_node;
   if  (TreeIsFull(ptree))
   {
      fprintf(stderr, "Дерево переполнено\n");
      return false;              /* преждевременный возврат      */
   }
   if (SeekItem(pi, ptree).child != NULL)
   {
      fprintf(stderr, "Попытка добавления дублированного элемента\n");
      return false;              /* преждевременный возврат      */
   }
   new_node = MakeNode(pi);      /* указывает на новый узел      */
   if (new_node == NULL)
   {
      fprintf(stderr, "Не удалось создать узел\n");
      return false;              /* преждевременный возврат      */
   }
   /* успешное создание нового узла */
   ptree->size++;
   if (ptree->root == NULL)      /* случай 1: дерево пустое      */
      ptree->root = new_node;    /* новый узел - корень дерева   */
   else                          /* случай 2: дерево не пустое   */
      AddNode(new_node,ptree->root); /* добавление узла к дереву */
   return true;                  /* возврат в случае успеха      */
}
```

Функции SeekItem(), MakeNode() и AddNode() не являются частью открытого интерфейса для типа Tree. Вместо этого они представляют собой статические функции, скрытые в файле tree.c, которые имеют дело с такими не относящимися к открытому интерфейсу деталями реализации, как узлы, указатели и структуры.

Функция MakeNode() довольно проста. Она обеспечивает динамическое выделение памяти и инициализацию узла. Аргументом функции является указатель на новый элемент, а ее возвращаемым значением – указатель на новый узел. Вспомните, что функция malloc() возвращает нулевой указатель, если она не может выделить запрошенную память. Функция MakeNode() инициализирует новый узел только в случае успешного выделения памяти. Вот код функции MakeNode():

```
static Trnode * MakeNode(const Item * pi)
{
   Trnode * new_node;
   new_node = (Trnode *) malloc(sizeof(Trnode));
   if (new_node != NULL)
   {
      new_node->item = *pi;
      new_node->left = NULL;
      new_node->right = NULL;
   }
   return new_node;
}
```

Функция AddNode() является второй по сложности в пакете двоичного дерева поиска. Она должна определить, куда должен быть помещен новый узел, и затем добавить его. В частности, ей необходимо сравнить новый элемент с корневым элементом, чтобы выяснить, в какое поддерево должен быть помещен новый элемент – левое или правое.

Если бы элемент был числом, то для выполнения сравнений можно было бы использовать операции < и >, а если бы строкой, то функцию strcmp(). Но элемент является структурой, содержащей две строки, так что для выполнения сравнений придется предусмотреть собственные функции. Функция ToLeft(), которая будет определена позже, возвращает значение True, если новый элемент должен быть помещен в левое поддерево, а функция ToRight() возвращает значение True, если новый элемент должен войти в правое поддерево. Эти две функции представляют собой аналоги операций < и >. Предположим, что новый элемент должен быть помещен в левое поддерево. Оно вполне может оказаться пустым. В таком случае функция просто устанавливает указатель на левый дочерний узел так, чтобы он ссылался на новый узел. А что, если левое поддерево не пустое? Тогда функция должна сравнить новый элемент с элементом в левом дочернем узле, чтобы выяснить, в какое поддерево дочернего узла должен быть помещен новый узел — левое или правое. Этот процесс должен продолжаться до тех пор, пока функция не достигнет пустого поддерева, в которое может быть добавлен новый узел. Один из возможных способов реализации такого поиска связан с рекурсией — применением функции AddNode() к дочернему, а не корневому узлу. Рекурсивная последовательность вызовов функции завершается, когда левое или правое поддерево оказывается пустым, т.е. когда root->left или root->right равно NULL. Имейте в виду, что root — это указатель на верхушку текущего поддерева, поэтому в каждом рекурсивном вызове он указывает на новое, расположенное на более низком уровне, поддерево. (Рекурсия обсуждалась в главе 9.)

```
static void AddNode (Trnode * new_node, Trnode * root)
{
   if (ToLeft(&new_node->item, &root->item))
   {
      if (root->left == NULL)             /* пустое поддерево,          */
         root->left = new_node;           /* поэтому добавить сюда узел */
      else
         AddNode(new_node, root->left);   /* иначе обработать поддерево */
   }
   else if (ToRight(&new_node->item, &root->item))
   {
      if (root->right == NULL)
         root->right = new_node;
      else
         AddNode(new_node, root->right);
   }
   else                                   /* дубликаты не допускаются   */
   {
      fprintf(stderr, "Ошибка местоположения в AddNode()\n");
      exit(1);
   }
}
```

Функции ToLeft() и ToRight() зависят от сущности типа Item. Члены клуба Nerfville Pet Club будут упорядочены в алфавитном порядке по кличкам. Если двое животных имеют одинаковые клички, они должны быть упорядочены по виду. Если их вид также совпадает, то два элемента являются дубликатами, что в базовом дереве поиска не допускается. Вспомните, что функция strcmp() из стандартной библиотеки С возвращает отрицательное число, если строка, представленная ее первым аргументом, предшествует строке во втором аргументе, ноль, если обе строки совпадают, и положительное число, если первая строка следует за второй. Функция ToRight() содержит аналогичный код. Использование этих двух функций вместо выполнения срав-

нений непосредственно в AddNode() упрощает адаптацию кода к новым требованиям. Вместо того чтобы переписывать функцию AddNode(), когда требуется другая форма сравнения, достаточно модифицировать функции ToLeft() и ToRight().

```
static bool ToLeft(const Item * i1, const Item * i2)
{
   int comp1;
   if ((comp1 = strcmp(i1->petname, i2->petname)) < 0)
      return true;
   else if (comp1 == 0 &&
            strcmp(i1->petkind, i2->petkind) < 0 )
      return true;
   else
      return false;
}
```

Поиск элемента

В трех функциях интерфейса — AddItem(), InTree() и DeleteItem() — предусмотрен поиск в дереве конкретного элемента. В рассматриваемой реализации для этого используется функция SeekItem(). С функцией DeleteItem() связано дополнительное требование: она должна знать родительский узел удаляемого элемента, чтобы дочерний указатель родительского узла можно было обновить, когда удаляется дочерний элемент. Таким образом, функция SeekItem() спроектирована так, чтобы возвращать структуру, содержащую два указателя: один указывает на узел, который содержит искомый элемент (NULL, если элемент не найден), а другой указывает на родительский узел (NULL, если данный узел является корневым и не имеет родительского узла). Тип структуры определен следующим образом:

```
typedef struct pair {
   Trnode * parent;
   Trnode * child;
} Pair;
```

Функцию SeekItem() можно реализовать рекурсивно. Однако чтобы ознакомить вас с разными приемами программирования, для нисходящего обхода дерева мы применим цикл while. Подобно AddNode(), для навигации по дереву функция SeekItem() использует ToLeft() и ToRight(). Первоначально SeekItem() устанавливает указатель look.child так, чтобы он ссылался на корень дерева, а затем, по мере прохода по пути к возможному местонахождению элемента, переустанавливает этот указатель на последующие поддеревья. Одновременно указатель look.parent устанавливается для ссылки на последующие родительские узлы. Если подходящего элемента не найдено, значением указателя look.child будет NULL. Если искомый элемент находится в корневом узле, look.parent равно NULL, т.к. корневой узел не имеет родительского узла. Ниже приведен код функции SeekItem().

```
static Pair SeekItem(const Item * pi, const Tree * ptree)
{
   Pair look;
   look.parent = NULL;
   look.child = ptree->root;
   if (look.child == NULL)
      return look;  /* преждевременный возврат */
   while (look.child != NULL)
   {
      if (ToLeft(pi, &(look.child->item)))
```

```
        {
           look.parent = look.child;
           look.child = look.child->left;
        }
        else if (ToRight(pi, &(look.child->item)))
        {
           look.parent = look.child;
           look.child = look.child->right;
        }
        else        /* если элемент не расположен ни слева,
                       ни справа, он должен быть таким же */
           break; /* look.child - это адрес узла, содержащего элемент */
     }
     return look;  /* возврат в случае успеха   */
}
```

Обратите внимание, что поскольку функция `SeekItem()` возвращает структуру, ее можно применять с операцией членства в структуре. Например, в функции `AddItem()` используется следующий код:

```
if (SeekItem(pi, ptree).child != NULL)
```

При наличии `SeekItem()` написание кода функции `InTree()` открытого интерфейса не составит труда:

```
bool InTree(const Item * pi, const Tree * ptree)
{
   return (SeekItem(pi, ptree).child == NULL) ? false : true;
}
```

Соображения по поводу удаления элемента

Удаление элемента представляет собой наиболее трудоемкую задачу, т.к. необходимо заново соединить остающиеся поддеревья для формирования допустимого дерева. Прежде чем приступить к программированию решения этой задачи, имеет смысл визуально представить действия, которые должны быть предприняты. На рис. 17.13 иллюстрируется простейший случай. Здесь удаляемый узел не имеет дочерних узлов.

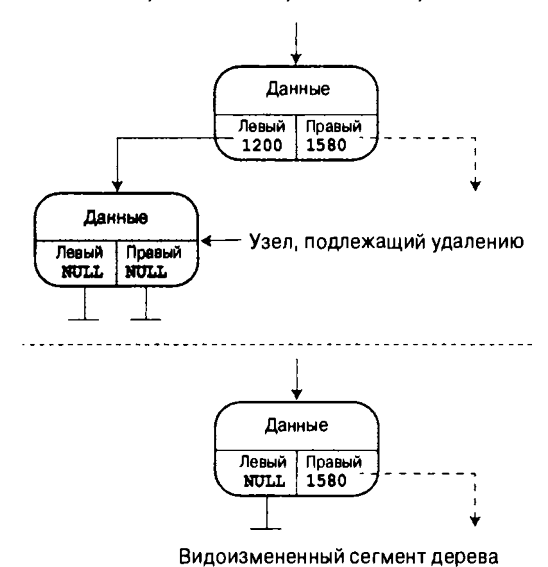


Рис. 17.13. Удаление листа

Такой узел называется *листом*. В этом случае понадобится только переустановить указатель в родительском узле в NULL и с помощью функции free() освободить память, занимаемую удаленным узлом.

Следующая по сложности задача – удаление узла с одним дочерним узлом. Удаление узла ведет к отделению дочернего поддерева от остального дерева. Для исправления такой ситуации адрес дочернего поддерева должен быть сохранен в родительском узле в позиции, которая ранее была занята адресом удаленного узла (рис. 17.14).

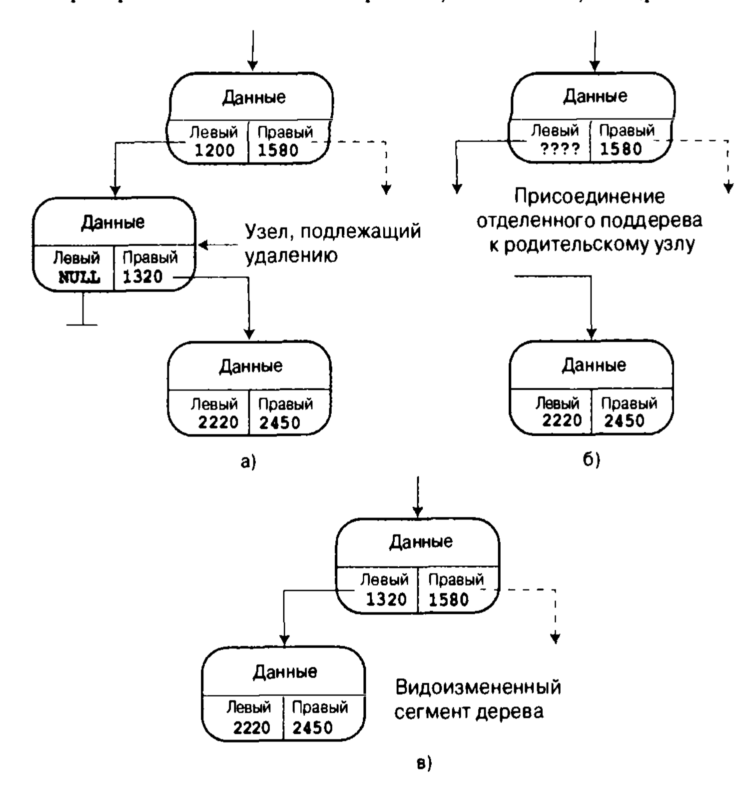


Рис. 17.14. Удаление узла с одним дочерним узлом

Последний случай связан с удалением узла, имеющего два поддерева. Одно поддерево, скажем, левое, может быть присоединено к тому узлу, к которому вначале был присоединен удаленный узел. Но куда поместить оставшееся поддерево? Вспомним базовый принцип формирования древовидной структуры. Каждый элемент в левом поддереве предшествует элементу в родительском узле, а каждый элемент в правом поддереве следует за элементом в родительском узле. Это означает, что каждый элемент в правом поддереве расположен в структуре дальше любого элемента из левого поддерева. Кроме того, поскольку правое поддерево ранее было частью поддерева, начинающегося с удаленного узла, каждый элемент в правом поддереве предшествует родительскому узлу удаленного узла. Вообразите себе спуск по дереву в поисках позиции для помещения начала правого поддерева. Он предшествует родительскому узлу, поэтому далее необходимо следовать вниз по левому поддереву. Однако начало поддерева должно быть расположено после всех элементов в левом поддереве, поэтому необходимо последовать правой ветвью левого поддерева и выяснить, имеется ли в ней место для нового узла. Если нет, потребуется продолжать спуск по правой ветви

левого поддерева до тех пор, пока свободное место не будет найдено. Этот подход продемонстрирован на рис. 17.15.

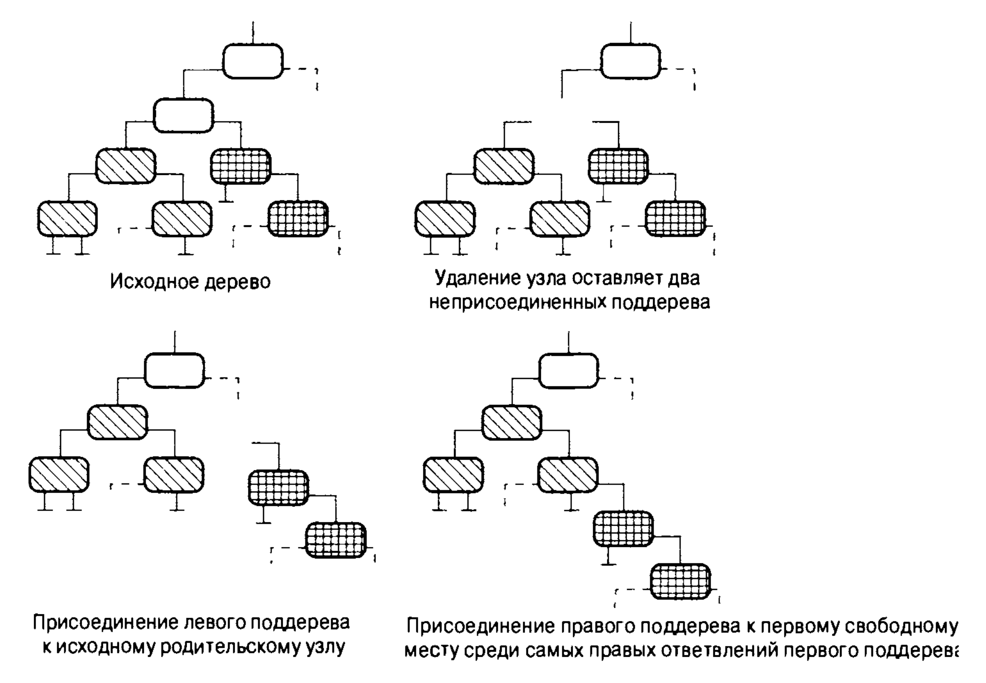


Рис. 17.15. Удаление узла с двумя дочерними узлами

Удаление узла

Теперь можно приступать к планированию необходимых функций, разделив работу на две задачи. Первая задача предусматривает связывание конкретного элемента с узлом, подлежащим удалению, а вторая заключается в действительном удалении узла. Следует отметить, что во всех случаях требуется модификация указателя в родительском узле, а это приводит к двум важным последствиям.

- Программа должна идентифицировать родительский узел удаляемого узла.
- Для изменения указателя код должен передавать функции удаления *адрес* этого указателя.

К первому моменту мы вернемся несколько позже, а пока проанализируем второй момент. Указатель, который нужно изменять, имеет тип `Trnode *`, т.е. является указателем на `Trnode`. Поскольку аргумент функции — адрес этого указателя, типом аргумента будет `Trnode **`, или указатель на указатель на `Trnode`. Предполагая, что подходящий адрес доступен, функцию удаления можно реализовать следующим образом:

```
static void DeleteNode(Trnode **ptr)
/* адрес родительского элемента, указывающего на целевой узел */
{
   Trnode * temp;
   if ( (*ptr)->left == NULL)
   {
```

```
        temp = *ptr;
        *ptr = (*ptr)->right;
        free(temp);
    }
    else if ( (*ptr)->right == NULL)
    {
        temp = *ptr;
        *ptr = (*ptr)->left;
        free(temp);
    }
    else   /* удаляемый узел имеет два дочерних узла */
    {
        /* выяснение места присоединения правого поддерева */
        for (temp = (*ptr)->left; temp->right != NULL;
             temp = temp->right)
           continue;
        temp->right = (*ptr)->right;
        temp = *ptr;
        *ptr =(*ptr)->left;
        free(temp);
    }
}
```

В этой функции явно обрабатываются три случая: узел без левого дочернего узла, узел без правого дочернего узла и узел с двумя дочерними узлами. Узел без дочерних узлов можно считать особым случаем узла без левого дочернего узла. Если узел не имеет левого дочернего узла, код присваивает адрес правого дочернего узла указателю на родительский узел. Но если узел не имеет также и правого узла, то значением этого указателя будет NULL, которое является полностью подходящим значением для случая узла без дочерних узлов.

Обратите внимание, что для отслеживания адреса удаляемого узла в коде применяется временный указатель. В результате переустановки родительского указателя (*ptr) программа утратила бы информацию о местоположении удаленного узла, а эта информация нужна для функции free(). Таким образом, исходное значение *ptr сохраняется в переменной temp, а затем используется для освобождения памяти, занимаемой удаленным узлом.

В коде для случая узла с двумя дочерними узлами вначале применяется указатель temp в цикле for для поиска свободного места в правой части левого поддерева. После его нахождения к нему присоединяется правое поддерево. Затем снова используется указатель temp для отслеживания местоположения удаленного узла. И, наконец, левое поддерево присоединяется к родительскому узлу, после чего узел, на который указывает temp, освобождается.

Обратите внимание, что поскольку ptr имеет тип Trnode **, то *ptr относится к типу Trnode *, делая его совпадающим по типу с указателем temp.

Удаление элемента

Оставшаяся нерешенной часть задачи касается связи узла с определенным элементом. Чтобы сделать это, можно воспользоваться функцией SeekItem(). Вспомните, что она возвращает структуру, содержащую указатель на родительский узел и указатель на узел, в котором находится элемент. Следовательно, указатель родительского узла можно применять для получения подходящего адреса и его передачи в функцию DeleteNode(). Такой план реализован в показанной ниже функции DeleteItem().

```
bool DeleteItem(const Item * pi, Tree * ptree)
{
   Pair look;

   look = SeekItem(pi, ptree);
   if (look.child == NULL)
      return false;

   if (look.parent == NULL)      /* удаление корневого элемента */
      DeleteNode(&ptree->root);
   else if (look.parent->left == look.child)
      DeleteNode(&look.parent->left);
   else
      DeleteNode(&look.parent->right);
   ptree->size--;

   return true;
}
```

Возвращаемое значение функции `SeekItem()` присваивается переменной `look` типа структуры. Если значение `look.child` равно `NULL`, поиск элемента безуспешен, и функция `DeleteItem()` завершает работу, возвращая `false`. Если элемент `Item` найден, функция обрабатывает три случая. Прежде всего, значение `NULL` переменной `look.parent` говорит о том, что элемент был найден в корневом узле. В таком случае родительский узел, который нужно было бы обновить, отсутствует. Вместо этого должен быть обновлен указатель `root` в структуре `Tree`. Следовательно, функция передает адрес этого указателя в функцию `DeleteNode()`. В противном случае код выясняет, в левом или правом дочернем узле родительского узла расположен удаляемый узел, и затем передает адрес соответствующего указателя.

Обратите внимание, что функция открытого интерфейса (`DeleteItem()`) оперирует понятиями, близкими конечному пользователю (элементами и деревьями), а скрытая функция `DeleteNode()` выполняет будничные действия с указателями.

Обход дерева

Обход дерева является более сложной задачей, чем обход связного списка, поскольку каждый узел имеет две ветви, по которым нужно проследовать. Такая природа ветвления делает естественным методом решения этой задачи рекурсию типа "разделяй и властвуй" (см. главу 9). В каждом узле функция должна выполнить следующие действия:

- обработать элемент в узле;
- обработать левое поддерево (рекурсивный вызов);
- обработать правое поддерево (рекурсивный вызов).

Данный процесс можно разбить на две функции: `Traverse()` и `InOrder()`. Обратите внимание, что функция `InOrder()` обрабатывает левое поддерево, затем элемент и после этого правое поддерево. Такая организация обработки приводит к обходу дерева в алфавитном порядке. При желании можете самостоятельно посмотреть, что происходит в случае использования других порядков обработки, скажем, "элемент, левое поддерево, правое поддерево" и "левое поддерево, правое поддерево, элемент".

```
void Traverse (const Tree * ptree, void (* pfun)(Item item))
{
   if (ptree != NULL)
      InOrder(ptree->root, pfun);
}
```

```
static void InOrder(const Trnode * root, void (* pfun)(Item item))
{
   if (root != NULL)
   {
      InOrder(root->left, pfun);
      (*pfun)(root->item);
      InOrder(root->right, pfun);
   }
}
```

Опустошение дерева

По существу опустошение дерева представляет собой тот же самый процесс, что и его обход. Другими словами, коду необходимо посетить каждый узел и применить к нему функцию `free()`. Код должен также переустановить члены структуры `Tree`, чтобы отразить пустое дерево. Функция `DeleteAll()` позаботится о структуре `Tree` и передаст задачу освобождения памяти функции `DeleteAllNodes()`. Последняя функция аналогична функции `InOrder()`. Она сохраняет значение указателя `root->right`, чтобы он оставался доступным после освобождения корня. Вот код упомянутых двух функций:

```
void DeleteAll(Tree * ptree)
{
   if (ptree != NULL)
      DeleteAllNodes(ptree->root);
   ptree->root = NULL;
   ptree->size = 0;
}

static void DeleteAllNodes(Trnode * root)
{
   Trnode * pright;

   if (root != NULL)
   {
      pright = root->right;
      DeleteAllNodes(root->left);
      free(root);
      DeleteAllNodes(pright);
   }
}
```

Завершенный пакет

Полный код файла `tree.c` представлен в листинге 17.11. Вместе с файлами `tree.h` и `tree.c` формируется программный пакет для древовидного представления.

Листинг 17.11. Файл реализации `tree.c`

```
/* tree.c -- функции поддержки дерева */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"

/* локальный тип данных */
typedef struct pair {
   Trnode * parent;
   Trnode * child;
} Pair;
```

```
/* прототипы локальных функций */
static Trnode * MakeNode(const Item * pi);
static bool ToLeft(const Item * i1, const Item * i2);
static bool ToRight(const Item * i1, const Item * i2);
static void AddNode (Trnode * new_node, Trnode * root);
static void InOrder(const Trnode * root, void (* pfun)(Item item));
static Pair SeekItem(const Item * pi, const Tree * ptree);
static void DeleteNode(Trnode **ptr);
static void DeleteAllNodes(Trnode * ptr);

/* определения функций */
void InitializeTree(Tree * ptree)
{
   ptree->root = NULL;
   ptree->size = 0;
}

bool TreeIsEmpty(const Tree * ptree)
{
   if (ptree->root == NULL)
      return true;
   else
      return false;
}

bool TreeIsFull(const Tree * ptree)
{
   if (ptree->size == MAXITEMS)
      return true;
   else
      return false;
}

int TreeItemCount(const Tree * ptree)
{
   return ptree->size;
}

bool AddItem(const Item * pi, Tree * ptree)
{
   Trnode * new_node;

   if  (TreeIsFull(ptree))
   {
      fprintf(stderr, "Дерево переполнено\n");
      return false;              /* преждевременный возврат      */
   }

   if (SeekItem(pi, ptree).child != NULL)
   {
      fprintf(stderr, "Попытка добавления дублированного элемента\n");
      return false;              /* преждевременный возврат      */
   }
   new_node = MakeNode(pi);      /* указывает на новый узел      */
   if (new_node == NULL)
   {
      fprintf(stderr, "Не удалось создать узел\n");
      return false;              /* преждевременный возврат      */
   }

   /* успешное создание нового узла */
   ptree->size++;
```

```
   if (ptree->root == NULL)             /* случай 1: дерево пустое     */
      ptree->root = new_node;           /* новый узел — корень дерева  */
   else                                 /* случай 2: дерево не пустое  */
      AddNode(new_node,ptree->root);    /* добавление узла к дереву    */
   return true;                         /* возврат в случае успеха     */
}
bool InTree(const Item * pi, const Tree * ptree)
{
   return (SeekItem(pi, ptree).child == NULL) ? false : true;
}
bool DeleteItem(const Item * pi, Tree * ptree)
{
   Pair look;
   look = SeekItem(pi, ptree);
   if (look.child == NULL)
      return false;
   if (look.parent == NULL)             /* удаление корневого элемента */
      DeleteNode(&ptree->root);
   else if (look.parent->left == look.child)
      DeleteNode(&look.parent->left);
   else
      DeleteNode(&look.parent->right);
   ptree->size--;

   return true;
}
void Traverse (const Tree * ptree, void (* pfun)(Item item))
{

   if (ptree != NULL)
      InOrder(ptree->root, pfun);
}
void DeleteAll(Tree * ptree)
{
   if (ptree != NULL)
      DeleteAllNodes(ptree->root);
   ptree->root = NULL;
   ptree->size = 0;
}
/* локальные функции */
static void InOrder(const Trnode * root, void (* pfun)(Item item))
{
   if (root != NULL)
   {
      InOrder(root->left, pfun);
      (*pfun)(root->item);
      InOrder(root->right, pfun);
   }
}
static void DeleteAllNodes(Trnode * root)
{
   Trnode * pright;
   if (root != NULL)
   {
      pright = root->right;
      DeleteAllNodes(root->left);
      free(root);
      DeleteAllNodes(pright);
   }
}
```

```
static void AddNode (Trnode * new_node, Trnode * root)
{
   if (ToLeft(&new_node->item, &root->item))
   {
      if (root->left == NULL)             /* пустое поддерево,           */
         root->left = new_node;           /* поэтому добавить сюда узел */
      else
         AddNode(new_node, root->left);   /* иначе обработать поддерево */
   }
   else if (ToRight(&new_node->item, &root->item))
   {
      if (root->right == NULL)
         root->right = new_node;
      else
         AddNode(new_node, root->right);
   }
   else                                   /* дубликаты не допускаются   */
   {
      fprintf(stderr, "Ошибка местоположения в AddNode()\n");
      exit(1);
   }
}

static bool ToLeft(const Item * i1, const Item * i2)
{
   int comp1;
   if ((comp1 = strcmp(i1->petname, i2->petname)) < 0)
      return true;
   else if (comp1 == 0 &&
          strcmp(i1->petkind, i2->petkind) < 0 )
      return true;
   else
      return false;
}

static bool ToRight(const Item * i1, const Item * i2)
{
   int comp1;
   if ((comp1 = strcmp(i1->petname, i2->petname)) > 0)
      return true;
   else if (comp1 == 0 &&
          strcmp(i1->petkind, i2->petkind) > 0 )
      return true;
   else
      return false;
}

static Trnode * MakeNode(const Item * pi)
{
   Trnode * new_node;
   new_node = (Trnode *) malloc(sizeof(Trnode));
   if (new_node != NULL)
   {
      new_node->item = *pi;
      new_node->left = NULL;
      new_node->right = NULL;
   }
   return new_node;
}
```

```
static Pair SeekItem(const Item * pi, const Tree * ptree)
{
    Pair look;
    look.parent = NULL;
    look.child = ptree->root;

    if (look.child == NULL)
        return look;  /* преждевременный возврат */

    while (look.child != NULL)
    {
        if (ToLeft(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->left;
        }

        else if (ToRight(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->right;
        }

        else /* если элемент не расположен ни слева, ни справа, он должен быть таким же */
            break;  /* look.child - это адрес узла, содержащего элемент */
    }

    return look;   /* возврат в случае успеха   */
}

static void DeleteNode(Trnode **ptr)
/* адрес родительского элемента, указывающего на целевой узел */
{
    Trnode * temp;

    if ( (*ptr)->left == NULL)
    {
        temp = *ptr;
        *ptr = (*ptr)->right;
        free(temp);
    }

    else if ( (*ptr)->right == NULL)
    {
        temp = *ptr;
        *ptr = (*ptr)->left;
        free(temp);
    }

    else   /* удаляемый узел имеет два дочерних узла */
    {
        /* выяснение места присоединения правого поддерева */
        for (temp = (*ptr)->left; temp->right != NULL;
              temp = temp->right)
            continue;
        temp->right = (*ptr)->right;
        temp = *ptr;
        *ptr =(*ptr)->left;
        free(temp);
    }
}
```

Тестирование пакета для древовидного представления

Теперь, когда реализации интерфейса и функций созданы, давайте применим их. В программе в листинге 17.12 используется меню с пунктами, предназначенными для добавления домашних животных в реестр членов клуба, вывода списка членов клуба, вывода количества членов, проверки членства и выхода из программы. Короткая функция main() сосредоточена на основной схеме программы. Большую часть работы выполняют поддерживающие функции.

Листинг 17.12. Программа petclub.c

```
/* petclub.c -- использование двоичного дерева поиска */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "tree.h"

char menu(void);
void addpet(Tree * pt);
void droppet(Tree * pt);
void showpets(const Tree * pt);
void findpet(const Tree * pt);
void printitem(Item item);
void uppercase(char * str);
char * s_gets(char * st, int n);

int main(void)
{
   Tree pets;
   char choice;

   InitializeTree(&pets);
   while ((choice = menu()) != 'q')
   {
      switch (choice)
      {
         case 'a' :  addpet(&pets);
            break;
         case 'l' :  showpets(&pets);
            break;
         case 'f' :  findpet(&pets);
            break;
         case 'n' :  printf("%d животных в клубе \n",
                              TreeItemCount(&pets));
            break;
         case 'd' :  droppet(&pets);
            break;
         default  :  puts("Ошибка в switch");
      }
   }
   DeleteAll(&pets);
   puts("Программа завершена.");

   return 0;
}

char menu(void)
{
   int ch;
```

```
    puts("Программа членства в клубе Nerfville Pet Club");
    puts("Введите букву, соответствующую вашему выбору:");
    puts("a) добавление животного  l) вывод списка животных");
    puts("n) количество животных   f) поиск животных");
    puts("d) удаление животного    q) выход");
    while ((ch = getchar()) != EOF)
    {
        while (getchar() != '\n')  /* отбросить оставшуюся часть строки */
            continue;
        ch = tolower(ch);
        if (strchr("alrfndq",ch) == NULL)
            puts("Введите букву a, l, f, n, d или q:");
        else
            break;
    }
    if (ch == EOF)       /* ввод символа EOF приводит к выходу из программы */
        ch = 'q';

    return ch;
}

void addpet(Tree * pt)
{
    Item temp;

    if (TreeIsFull(pt))
        puts("В клубе больше нет мест!");
    else
    {
        puts("Введите кличку животного:");
        s_gets(temp.petname,SLEN);
        puts("Введите вид животного:");
        s_gets(temp.petkind,SLEN);
        uppercase(temp.petname);
        uppercase(temp.petkind);
        AddItem(&temp, pt);
    }
}

void showpets(const Tree * pt)
{
    if (TreeIsEmpty(pt))
        puts("Записи отсутствуют!");
    else
        Traverse(pt, printitem);
}

void printitem(Item item)
{
    printf("Животное: %-19s  Вид: %-19s\n", item.petname,
           item.petkind);
}

void findpet(const Tree * pt)
{
    Item temp;

    if (TreeIsEmpty(pt))
    {
        puts("Записи отсутствуют!");
        return;     /* если дерево пустое, выйти из функции */
    }
```

```
    puts("Введите кличку животного, которое хотите найти:");
    s_gets(temp.petname, SLEN);
    puts("Введите вид животного:");
    s_gets(temp.petkind, SLEN);
    uppercase(temp.petname);
    uppercase(temp.petkind);
    printf("%s по имени %s ", temp.petkind, temp.petname);
    if (InTree(&temp, pt))
        printf("является членом клуба.\n");
    else
        printf("не является членом клуба.\n");
}

void droppet(Tree * pt)
{
    Item temp;

    if (TreeIsEmpty(pt))
    {
        puts("Записи отсутствуют!");
        return;     /* если дерево пустое, выйти из функции */
    }

    puts("Введите кличку животного, которое нужно исключить из клуба:");
    s_gets(temp.petname, SLEN);
    puts("Введите вид животного:");
    s_gets(temp.petkind, SLEN);
    uppercase(temp.petname);
    uppercase(temp.petkind);
    printf("%s по имени %s ", temp.petkind, temp.petname);
    if (DeleteItem(&temp, pt))
        printf("исключен(а) из клуба.\n");
    else
        printf("не является членом клуба.\n");
}

void uppercase(char * str)
{
    while (*str)
    {
        *str = toupper(*str);
        str++;
    }
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');     // поиск новой строки
        if (find)                    // если адрес не равен NULL,
            *find = '\0';            // поместить туда нулевой символ
        else
            while (getchar() != '\n')
                continue;            // отбросить остаток строки
    }
    return ret_val;
}
```

Программа преобразует все буквы в прописные, поэтому *СНАФФИ*, *Снаффи* и *снаффи* не считаются разными кличками. Ниже показаны результаты пробного запуска.

```
Программа членства в клубе Nerfville Pet Club
Введите букву, соответствующую вашему выбору:
a) добавление животного   l) вывод списка животных
n) количество животных    f) поиск животных
d) удаление животного     q) выход
a
Введите кличку животного:
Куинси
Введите вид животного:
свинья
Программа членства в клубе Nerfville Pet Club
Введите букву, соответствующую вашему выбору:
a) добавление животного   l) вывод списка животных
n) количество животных    f) поиск животных
d) удаление животного     q) выход
a
Введите кличку животного:
Бенни Ха-ха
Введите вид животного:
попугай
Программа членства в клубе Nerfville Pet Club
Введите букву, соответствующую вашему выбору:
a) добавление животного   l) вывод списка животных
n) количество животных    f) поиск животных
d) удаление животного     q) выход
a
Введите кличку животного:
Дон Базилио
Введите вид животного:
домашний кот
Программа членства в клубе Nerfville Pet Club
Введите букву, соответствующую вашему выбору:
a) добавление животного   l) вывод списка животных
n) количество животных    f) поиск животных
d) удаление животного     q) выход
n
3 животных в клубе
Программа членства в клубе Nerfville Pet Club
Введите букву, соответствующую вашему выбору:
a) добавление животного   l) вывод списка животных
n) количество животных    f) поиск животных
d) удаление животного     q) выход
l
Животное: БЕННИ ХА-ХА         Вид: ПОПУГАЙ
Животное: ДОН БАЗИЛИО         Вид: ДОМАШНИЙ КОТ
Животное: КУИНСИ              Вид: СВИНЬЯ
Программа членства в клубе Nerfville Pet Club
Введите букву, соответствующую вашему выбору:
a) добавление животного   l) вывод списка животных
n) количество животных    f) поиск животных
d) удаление животного     q) выход
q
Программа завершена.
```

Соображения по поводу дерева

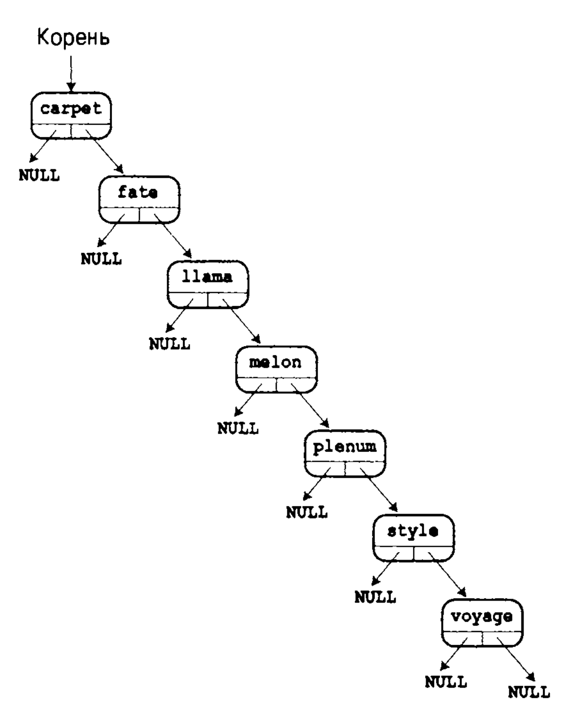


Рис. 17.16. Плохо сбалансированное двоичное дерево поиска

Двоичное дерево поиска обладает рядом недостатков. Например, двоичное дерево поиска эффективно, только если оно полностью заполнено, или *сбалансировано*. Предположим, что вы сохраняете слова, которые вводятся в произвольном порядке. Есть шансы, что результирующее дерево будет иметь довольно небольшую глубину, как на рис. 17.12. Но представим, что вводятся данные в алфавитном порядке. Тогда каждый новый узел будет добавляться справа, и дерево может приобрести вид, показанный на рис. 17.16. Дерево на рис. 17.12 называют *сбалансированным*, а на рис. 17.16 – *несбалансированным*. Поиск в несбалансированном дереве не является более эффективным, чем последовательный поиск в связном списке.

Один из способов избежать получения вытянутых деревьев – применять более тщательный подход к их построению. Если дерево или поддерево начинает становиться слишком перекошенным в одну из сторон, необходимо реорганизовать узлы для улучшения сбалансированности дерева. Аналогично реорганизация дерева может требоваться после удаления узлов. Математики Г.М. Адельсон-Вельский и Е.М. Ландис разработали для этого алгоритм. Деревья, построенные их методом, называют *АВЛ-деревьями* (от начальных букв их фамилий). Построение сбалансированного дерева занимает больше времени, т.к. должны предприниматься дополнительные действия по реструктуризации, но тем самым обеспечивается максимальная или близкая к максимальной эффективность поиска.

Иногда может требоваться двоичное дерево поиска, которое допускает наличие дублированных элементов. Для примера предположим, что нужно проанализировать какой-то текст, отслеживая количество появлений в нем каждого слова. Один из возможных подходов к решению этой задачи предусматривает определение `Item` как структуры, которая содержит одно слово и его количество. Когда слово встречается в тексте первый раз, оно добавляется в дерево, а количество устанавливается в `1`. При следующем появлении этого же слова программа находит содержащий его узел и инкрементирует количество. Преобразование базового двоичного дерева поиска, чтобы оно работало в подобной манере, не займет много времени.

Для ознакомления с еще одним возможным вариантом дерева, мы снова обратимся к примеру с клубом Nelville Pet Club. В этом примере сортировка дерева осуществлялась как по кличкам, так и по видам животных. Поэтому оно могло бы содержать кота Сэма в одном узле, собаку Сэма – во втором и хомяка Сэма – в третьем. Тем не менее, в дереве не могло быть двух котов с кличкой Сэм. Еще один возможный под-

ход заключается в упорядочении дерева только по кличкам животных. Само по себе это изменение разрешило бы существование только одного Сэма, независимо от вида, но затем можно было бы определить `Item` как список структур, а не как одиночную структуру. Тогда при первом появлении животного с кличкой Салли программа создала бы новый узел, затем новый список, после чего добавила бы Салли и ее вид в этот список. Следующее животное Салли было бы направлено в этот же узел и добавлено в список.

Дополнительные библиотеки

Вероятно, вы уже поняли, что реализация абстрактных типов данных, таких как связный список или дерево, является трудоемкой задачей, сопряженной с множеством потенциальных ошибок. Дополнительные библиотеки предлагают альтернативный подход: позволить выполнить всю работу и тестирование кому-то другому. Ознакомившись с двумя сравнительно простыми примерами в этой главе, вы сможете гораздо лучше понимать и ценить значимость таких библиотек.

Другие направления

В этой книге были раскрыты важные функциональные возможности языка C, но мы лишь вскользь затронули библиотеку. Библиотека ANSI C содержит множество полезных функций. Большинство реализаций также предоставляют обширные библиотеки, специфичные для конкретных систем. Компиляторы для Windows поддерживают графический интерфейс Windows. Компиляторы C для Macintosh предлагают функции для доступа к инструментальному набору Macintosh, облегчающему создание программ со стандартным интерфейсом Macintosh или программ для систем iOS, таких как iPhone и iPad. Аналогично имеются инструменты для построения программ Linux с графическими интерфейсами. Найдите время для исследования возможностей, которые поддерживает ваша система. Если того, что вам нужно, в системе отсутствует, создайте собственные функции. Возможность их создания — неотъемлемая часть C. Если вы думаете, что можете создать, скажем, более совершенную функцию ввода — сделайте это! И по мере совершенствования и оттачивания своих навыков программирования вы сможете перейти от написания простого кода C к получению блестящего кода C.

Если концепции списков, очередей и деревьев показались вам интересными и полезными, можете почитать соответствующие книги или прослушать курс по более сложным технологиям программирования. Ученые в области компьютерных вычислений тратят массу энергии и таланта на разработку и анализ алгоритмов и способов представления данных. Возможно, кто-то уже разработал именно то средство, в котором вы нуждаетесь.

После освоения языка C можете заняться изучением C++, Objective C или Java. Эти *объектно-ориентированные* языки произрастают из C. Язык C уже содержит объекты данных, варьирующиеся по сложности от простой переменной типа `char` до крупных и сложных структур. Объектно-ориентированные языки развивают идею объектов еще больше. Например, свойства объекта определяют не только то, какие виды информации он может хранить, но также и разновидности операций, которые могут над ним выполняться. Описанные в настоящей главе абстрактные типы данных соответствуют этому подходу. Кроме того, объекты могут наследовать свойства от других объектов. Объектно-ориентированное программирование переносит концепцию модульности на более высокий уровень абстракции, чем это имеет место в языке C, и применяется при разработке больших программ.

Перечень дополнительных книг, которые могут вас заинтересовать, приведен в разделе I приложения Б.

Ключевые понятия

Тип данных характеризуется способами структурирования и хранения данных, а также возможными операциями над ними. Абстрактный тип данных (abstract data type – ADT) абстрактным образом определяет свойства и операции, характеризующие тип. Концептуально тип ADT можно преобразовать в код на конкретном языке программирования в два этапа. Первый этап связан с определением программного интерфейса. На языке C это можно сделать за счет использования заголовочного файла для определения имен типов и объявления прототипов функций, соответствующих допустимым операциям. Второй этап состоит в реализации интерфейса. На языке C это можно сделать в виде файла исходного кода, который предоставляет определения функций, соответствующих прототипам.

Резюме

Список, очередь и двоичное дерево являются примерами абстрактных типов данных, обычно применяемых в программировании. Их часто реализуют посредством динамического выделения памяти и связанных структур, но иногда их лучше реализовать с помощью массивов.

Если в программе используется конкретный тип (скажем, очередь или дерево), ее следует писать в терминах интерфейса типа. Это позволит модифицировать и совершенствовать реализацию типа, не изменяя программы, в которых применяется интерфейс типа.

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля приведены в приложении А.

1. Что требуется для определения типа данных?

2. Почему обход связного списка, приведенного в листинге 17.2, может осуществляться только в одном направлении? Как можно было бы изменить определение `struct film`, чтобы обход списка стал возможен в обоих направлениях?

3. Что такое ADT?

4. Функция `QueueIsEpty()` принимает в качестве аргумента указатель на структуру `queue`, но ее можно было бы написать так, чтобы она принимала саму структуру `queue`, а не указатель на нее. Каковы преимущества и недостатки каждого из этих подходов?

5. *Стек* является еще одной формой данных из семейства списков. В стеке добавления и удаления могут выполняться только с одной стороны списка. Говорят, что элементы "заталкиваются" в стек и "выталкиваются" из него. Следовательно, стек представляет собой структуру LIFO (*last in, first out* – "последним прибыл, первым обслужен").

   а. Определите тип ADT для стека.

   б. Определите программный интерфейс стека, т.е. заголовочный файл `stack.h`.

6. Каково максимальное количество сравнений, которые требуются при последовательном поиске и двоичном поиске для определения того, что конкретный элемент отсутствует в упорядоченном списке из 3 элементов? В списке из 1023 элементов? В списке из 65 535 элементов?

7. Предположим, что программа создает двоичное дерево поиска слов с использованием алгоритма, описанного в этой главе. Нарисуйте дерево, исходя из предположения, что слова были введены в следующем порядке:

   а. nice food roam dodge gate office wave

   б. wave roam office nice gate food dodge

   в. food dodge roam wave office gate nice

   г. nice roam office food wave gate dodge

8. Взгляните на двоичные деревья, созданные при ответе на вопрос для самоконтроля под номером 7. Как они будут выглядеть после удаления из них слова *food* с помощью алгоритма, описанного в этой главе?

Упражнения по программированию

1. Модифицируйте код в листинге 17.2 так, чтобы он отображал список фильмов как в исходном, так и в обратном порядке. Один из возможных подходов предусматривает изменение определения связного списка для обеспечения обхода списка в обоих направлениях. Другой подход заключается в применении рекурсии.

2. Предположим, что в файле `list.h` (листинг 17.3) используется следующее определение списка:

```
typedef struct list
{
    Node * head;  /* указывает на начало списка */
    Node * end;   /* указывает на конец списка  */
} List;
```

   Перепишите функции в файле `list.c` (листинг 17.5), чтобы они соответствовали этому определению, и протестируйте результирующий код с помощью программы `films3.c` (листинг 17.4).

3. Предположим, что в файле `list.h` (листинг 17.3) используется следующее определение списка:

```
#define MAXSIZE 100
typedef struct list
{
    Item entries[MAXSIZE];  /* массив элементов              */
    int items;              /* количество элементов в списке */
} List;
```

   Перепишите функции в файле `list.c` (листинг 17.5), чтобы они соответствовали этому определению, и протестируйте результирующий код с помощью программы `films3.c` (листинг 17.4).

4. Перепишите программу `mall.c` (листинг 17.7), чтобы она моделировала киоск с двумя окошками и двумя очередями.

5. Напишите программу, которая позволяет ввести строку. Программа затем должна заталкивать в стек символы строки по одному (см. вопрос для самоконтроля под номером 5), выталкивать символы из стека и, наконец, отображать их. В результате символы отображаются в обратном порядке.

6. Напишите функцию, которая принимает три аргумента: имя отсортированного массива целых чисел, количество элементов в массиве и целое число, которое нужно найти. Функция возвращает значение 1, если целое число присутствует в массиве, и 0 — если отсутствует. Воспользуйтесь двоичным поиском.

7. Напишите программу, которая открывает и считывает текстовый файл, фиксируя количество появлений в нем каждого слова. Используйте двоичное дерево поиска, модифицированное для хранения слова и количества его повторений. После того как программа прочитает файл, она должна отобразить меню, состоящее из трех пунктов. Первый пункт приводит к выводу списка всех слов с указанием их повторений. Второй обеспечивает возможность ввода слова, а программа должна сообщить количество вхождений этого слова в файле. Результатом третьего пункта меню должен быть выход из программы.

8. Модифицируйте программу для клуба любителей животных, чтобы все животные с одинаковыми кличками хранились в одном и том же узле списка. Когда пользователь выбирает поиск животного, программа должна запросить кличку животного, после чего вывести список всех животных (вместе с их видами), имеющих данную кличку.

A

Ответы на вопросы для самоконтроля

Ответы на вопросы для самоконтроля из главы 1

1. Полностью переносимая программа – это программа, код которой без каких-либо изменений может быть скомпилирован в программу, успешно выполняемую на широком разнообразии компьютерных систем.
2. Файл исходного кода содержит код в том виде, в каком он написан на языке, который использует программист. Файл объектного кода содержит код на машинном языке; ему не обязательно быть полным кодом завершенной программы. Исполняемый файл содержит полный код на машинном языке, формирующий исполняемую программу.
3. а. Определение целей программы.
   б. Проектирование программы.
   в. Написание кода программы.
   г. Компиляция программы.
   д. Запуск программы.
   е. Тестирование и отладка программы.
   ж. Сопровождение и модификация программы.
4. Компилятор транслирует исходный код (например, код, написанный на языке С) в эквивалентный код на машинном языке, называемый также *объектным кодом*.
5. Компоновщик объединяет исходный код с кодом библиотек и кодом запуска для создания исполняемой программы.

Ответы на вопросы для самоконтроля из главы 2

1. Они называются функциями.
2. Синтаксическая ошибка – это нарушение правил, регламентирующих составление предложений или программ. Примером ошибки синтаксиса русского языка может служить предложение: "Говорить моя хорошо по-русски". Ниже приведен пример синтаксической ошибки в программе на языке С:
   ```
   printf"Куда подевались скобки?";
   ```
3. Семантическая ошибка – это ошибка, связанная с неправильным применением понятий. Например: "Это предложение – прекрасный образец английского языка". Или в программе на языке С (имя `thrice_n` по смыслу означает "трижды `n`"):
   ```
   thrice_n=3 + n;
   ```
4. Строка 1: начните строку с символа #; правильно введите имя файла `stdio.h`; поместите это имя файла в угловые скобки.

   Строка 2: используйте `()`, а не `{}`; завершите комментарий символами `*/`, а не `/*`.

   Строка 3: используйте `{`, а не `(`.

   Строка 4: дополните оператор символом точки с запятой.

Строка 5: только эта строка (пустая) в программе является правильной!

Строка 6: используйте для присваивания символ =, а не :=. (Судя по всему, Джо немного знаком с языком Pascal.) Для количества недель в году укажите 52, а не 56.

Строка 7: эта строка должна выглядеть следующим образом:

```
printf("В году содержится %d недель.\n", s);
```

Строка 9: эта строка отсутствует, но должна существовать и состоять из закрывающей фигурной скобки }.

После внесения всех исправлений код должен выглядеть следующим образом:

```
#include <stdio.h>
int main(void) /* выводит количество недель в году */
{
    int s;

    s = 52;
    printf("В году содержится %d недель.\n", s);
    return 0;
}
```

5. а. `Бе, бе, Черная Овечка.У тебя найдется шерсть для меня?`

   (Обратите внимание на отсутствие пробела после точки. Пробел можно вставить, указав " У вместо "У.)

   б. `Прочь!`
   `Вот наглая свинья!`

   (Обратите внимание на то, что курсор остается в конце второй строки.)

   в. `Что?`
   `Не/nклюет?`

   (Обратите внимание, что обычная косая черта (/) не оказывает такое же влияние, как обратная косая черта (\); она просто выводится на экран.)

   г. `2 + 2 = 4`

   (Обратите внимание на замену каждой последовательности символов %d значением соответствующей переменной из списка. Кроме того, символ + означает операцию сложения, и это вычисление может выполняться внутри оператора printf().)

6. int и char (main — имя функции, function (функция) — технический термин, относящийся к языку С, а = представляет собой символ операции.)

7. `printf("Текст содержал %d слов и %d строк.\n", words, lines);`

8. После выполнения строки 7 значение a равно 5, a b — 2. После выполнения строки 8 значения a и b равны 5. После выполнения строки 9 значения a и b по-прежнему равны 5. (Обратите внимание, что значение a не может быть равным 2, поскольку на момент выполнения оператора a = b; значение переменной b уже изменено на 5.)

9. После выполнения строки 7 значение x равно 10, а значение b — 5. После выполнения строки 8 значение x равно 10, а значение y — 15. После выполнения строки 9 значение x равно 150, а значение y — 15.

Ответы на вопросы для самоконтроля из главы 3

1. а. int, возможно short, unsigned или unsigned short; население города представляет собой целое число.

   б. float; маловероятно, чтобы стоимость была целым числом. (Можно было бы использовать тип double, но в данном случае повышенная точность не требуется.)

   в. char.

   г. int, возможно unsigned.

2. Во-первых, тип long может вмещать большие числа, чем int; во-вторых, если требуется обработка больших значений, то использование типа, для которого во всех системах гарантируется длина, по меньшей мере, 32 бита, улучшает переносимость кода.

3. Чтобы получить в точности 32 бита, можно использовать тип int32_t, если он определен для данной системы. Для более короткого типа, который мог бы хранить, по меньшей мере, 32 бита, необходимо применять int_least32_t. Чтобы получить тип, который обеспечил бы самые быстрые вычисления с 32-битными значениями, следует выбрать int_fast32_t.

4. а. Константа char (но сохраненная как значение типа int).

   б. Константа double.

   в. Константа unsigned int, представленная в шестнадцатеричном формате.

   г. Константа double.

5. Строка 1: должна иметь вид #include <stdio.h>.

   Строка 2: должна иметь вид int main(void).

   Строка 3: необходимо использовать {, а не (.

   Строка 4: между g и h должна находиться запятая, а не точка с запятой.

   Строка 6 (пустая): в порядке.

   Строка 7: перед е должна находиться хотя бы одна цифра. В данном случае вполне подошло бы 1e21 или 1.0e21, хотя они достаточно большие.

   Строка 8: в порядке, во всяком случае, с точки зрения синтаксиса.

   Строка 9: необходимо применять }, а не).

   Отсутствующие строки. Во-первых, переменной rate никогда не присваивается значение. Во-вторых, переменная h никогда не используется. Кроме того, программа совершенно не информирует пользователя о результатах вычислений. Ни одна из этих ошибок не помешает запуску программы (хотя может быть выведено предупреждение о неиспользуемой переменной), но они преуменьшают и без того ограниченную ее пригодность. Кроме того, в конце программы должен быть предусмотрен оператор return.

   Ниже представлена одна из возможных корректных версий:

```
#include <stdio.h>
int main(void)
{
    float g, h;
    float tax, rate;
```

```
    rate = 0.08;
    g = 1.0e5;
    tax = rate*g;
    h = g + tax;
    printf("Вы должны $%f плюс $%f налогов, что всего составляет $%f.\n",
           g, tax, h);
    return 0;
}
```

6.

	Константа	Тип	Спецификатор
а.	12	int	%d
б.	0X3	unsigned int	%#X
в.	'C'	char (в действительности int)	%c
г.	2.34E07	double	%e
д.	'\040'	char (в действительности int)	%c
е.	7.0	double	%f
ж.	6L	long	%ld
з.	6.0f	float	%f
и.	0x5.b6p12	float	%a

7.

	Константа	Тип	Спецификатор
а.	12	unsigned int	%#o
б.	2.9e05L	long double	%Le
в.	's'	char (в действительности int)	%c
г.	100000	long	%ld
д.	'\n'	char (в действительности int)	%c
е.	20.0f	float	%f
ж.	0x44	unsigned int	%x

8.
```
printf("Шансы попасть в %d были %ld к 1.\n", imate, shot);
printf("Счет %f не соответствует уровню %c.\n", log, grade);
```

9.
```
ch = '\r';
ch = 13;
ch = '\015'
ch = '\xd'
```

10. Строка 0: должна содержать `#include <stdio.h>`.

Строка 1: необходимо использовать `/*` и `*/` либо `//`.

Строка 3: должно быть `int cows, legs;`

Строка 4: должно быть `насчитали?\n");`

Строка 5: `%d`, а не `%c`; следует поменять `legs` на `&legs`.

Строка 7: `%d`, а не `%f`.

Кроме того, нужно добавить оператор `return`.

Вот одна из возможных корректных версий:

```
#include <stdio.h>
int main(void) /* эта программа безупречна */
{
   int cows, legs;
   printf("Сколько коровьих ног вы насчитали?\n");
   scanf("%d", &legs);
   cows = legs / 4;
   printf("Отсюда следует, что есть %f коров(а)", cows);
   return 0;
}
```

11. а. Символ новой строки.

   б. Символ обратной косой черты.

   в. Символ двойной кавычки.

   г. Символ табуляции.

Ответы на вопросы для самоконтроля из главы 4

1. Программа функционирует некорректно. Первый оператор `scanf()` читает только имя, оставляя фамилию незатронутой, но по-прежнему находящейся в буфере ввода. (Этот буфер представляет собой просто временную область, используемую для хранения входных данных.) Когда следующий оператор `scanf()` переходит к считыванию веса, он продолжает чтение с того места, где была завершена предыдущая попытка, и пытается прочитать фамилию как значение веса. Это ведет к ошибке работы `scanf()`. С другой стороны, если в ответ на запрос имени ввести что-то вроде `Иван 144`, то `144` будет применяться в качестве значения веса, несмотря на то, что оно введено до выдачи запроса на ввод веса.

2. а. `Он продал эту картину за $234.50`

   б. Hi!

   (Примечание: первый символ – это символьная константа, второй – десятичное целое значение, преобразованное в символ, а третий – восьмеричное ASCII-представление символьной константы.)

   в.
   ```
   Его Гамлет был хорош, и без намека на вульгарность.
   содержит 51 символов.
   ```

   г. `Является ли 1.20e+003 тем же, что и 1201.00?`

3. Необходимо использовать символы `\"`, как показано ниже:

```
printf("\"%s\"\nсодержит %d символов.\n", Q, strlen(Q));
```

4. Корректная версия выглядит следующим образом:

```
#include <stdio.h>    /* не забудьте включить эту строку*/
#define B "booboo"    /* добавьте символы # и кавычки   */
#define X 10          /* добавьте символ #              */
int main(void)        /* вместо main(int)               */
{
   int age;
   int xp;            /* объявите все переменные        */
```

```
    char name[40];      /* создайте массив                  */
    printf("Введите свое имя.\n"); /* вставьте символ \n для улучшения
                                       читабельности */
    scanf("%s", name);
    printf("Хорошо, %C, а сколько вам лет?\n", name); /* %s для строки */
    scanf("%d", &age);        /* %d, а не %f, &age, а не age */
    xp = age + X;
    printf("Неужели, %s! Вам должно быть, по меньшей мере, %d.\n", B, xp);
    return 0;              /* не перезапускать            */
}
```

5. Вспомните о конструкции `%%`, предназначенной для вывода символа `%`.

   ```
   printf("Данный экземпляр книги \"%s\" стоит $%0.2f.\n", BOOK, cost);
   printf("Это %0.0f%% от цены в прайс-листе.\n", percent);
   ```

6. а. `%d`

   б. `%4X`

   в. `%10.3f`

   г. `%12.2e`

   д. `%-30s`

7. а. `%15lu`

   б. `%#4x`

   в. `%-12.2E`

   г. `%+10.3f`

   д. `%8.8s`

8. а. `%6.4d`

   б. `%*o`

   в. `%2c`

   г. `%+0.2f`

   д. `%-7.5s`

9. а.
   ```
   int dalmations;
   scanf("%d", &dalmations);
   ```

   б.
   ```
   float kgs, share;
   scanf("%f%f", &kgs, &share);
   ```

   (Примечание: в функции ввода спецификаторы формата `e`, `f` и `g` можно использовать взаимозаменяемо. Кроме того, для всех спецификаторов кроме `%c` наличие пробелов между спецификаторами преобразования никакой роли не играет.)

   в.
   ```
   char pasta[20];
   scanf("%s", pasta);
   ```

   г.
   ```
   char action[20];
   int value;
   scanf("%s %d", action, &value);
   ```

   д.
   ```
   int value;
   scanf("%*s %d", &value);
   ```

10. Пробельные символы – это символы пробела, табуляции и новой строки. В языке С пробельные символы служат для отделения конструкций друг от друга; в scanf() пробелы используются для разделения последовательных элементов ввода.

11. Символ z в %z является модификатором, а не спецификатором, поэтому он требует указания спецификатора для модификации. Вы могли бы использовать %zd для вывода результата по основанию 10 или другого спецификатора для вывода в системе счисления с другим основанием, например, %zx для шестнадцатеричной формы.

12. Были бы выполнены подстановки. К сожалению, препроцессор не в состоянии различать, какие фигурные скобки должны быть заменены круглыми, а какие нет. Таким образом, программа

```
#define ( {
#define ) }
int main(void)
(
   printf("Привет, Великан!\n");
)
```

превратилась бы в

```
int main{void}
{
   printf{"Привет, Великан!\n"};
}
```

Ответы на вопросы для самоконтроля из главы 5

1. а. 30.

   б. 27 (а не 3). Выражение (12 + 6)/(2*3) в результате дало бы 3.

   в. x = 1, y = 1 (целочисленное деление).

   г. x = 3 (целочисленное деление) и y = 9.

2. а. 6 (сводится к 3 + 3.3).

   б. 52.

   в. 0 (сводится к 0 * 22.0).

   г. 13 (сводится к 66.0 / 5, или 13.2, а затем присваивается переменной типа int).

3. а. 37.5 (сводится к 7.5 * 5.0)

   б. 1.5 (сводится к 30.0 / 20.0)

   в. 35 (сводится к 7 * 5)

   г. 37 (сводится к 150 / 4)

   д. 37.5 (сводится к 7.5 * 5)

   е. 35.0 (сводится к 7 * 5.0)

4. Строка 0: необходимо включить <stdio.h>.

   Строка 3: должна заканчиваться точкой с запятой, а не запятой.

Строка 6: оператор `while` образует бесконечный цикл, поскольку значение `i` остается равным 1 и всегда меньше 30. Вероятно, намерения были записать `while(i++ < 30)`.

Строки 6–8: судя по отступам, строки 7 и 8 должны были образовывать блок, однако отсутствие фигурных скобок означает, что цикл `while` включает в себя только строку 7. Необходимо добавить фигурные скобки.

Строка 7: поскольку и 1, и `i` – целые числа, результат деления будет равен 1 при `i = 1` и 0 при всех более высоких значениях. Использование выражения `n = 1.0/i;` привело бы к преобразованию `i` в тип с плавающей запятой перед выполнением операции деления, и общий результат оказался бы ненулевым.

Строка 8: в управляющем операторе опущен символ новой строки (`\n`). Это приводит к тому, что числа выводятся в одной строке, когда такое возможно.

Строка 10: должна содержать `return 0;`

Вот скорректированная версия:

```
#include <stdio.h>
int main(void)
{
   int i = 1;
   float n;

   printf("Будьте внимательны! Далее идет последовательность дробей!\n");
   while (i++ < 30)
   {
     n = 1.0/i;
     printf(" %f\n", n);
   }
   printf("На этом все!\n");
   return 0;
}
```

5. Основная проблема кроется во взаимодействии оператора проверки условия (является ли значение `sec` больше 0) и оператором `scanf()`, который получает значение переменной `sec`. В частности, при первом выполнении проверки условия программа не имеет ни малейшей возможности получить значение для `sec`, и сравнение будет выполняться со случайным значением, которое оказалось в используемой ячейке памяти; оно может быть больше 0, а может и не быть. Одно (хотя и не очень изящное) решение предусматривает инициализацию `sec`, скажем, 1, чтобы проверка условия проходила в первый раз. Это вскрывает вторую проблему. Когда вы, в конце концов, вводите 0, чтобы остановить программу, значение `sec` проверяется только после завершения цикла, и происходит вывод результатов для 0 секунд. В действительности в программе требуется оператор `scanf()`, который бы выполнялся перед проверкой условия оператора `while`. Этого можно добиться, изменив центральную часть программы следующим образом:

```
scanf("%d", &sec);
while ( sec > 0 ) {
   min = sec/S_TO_M;
   left = sec % S_TO_M;
   printf("%d секунд - это %d минут %d секунд. \n", sec, min, left);
   printf("Следующее значение?\n");
   scanf("%d", &sec);
}
```

Сначала выполняется оператор scanf(), находящийся снаружи цикла, а затем оператор scanf() в конце цикла (следовательно, прямо перед началом новой итерации цикла). Это распространенный метод решения проблем подобного рода, и именно поэтому он был применен в листинге 5.9.

6. Вывод программы имеет следующий вид:

```
%s! C is cool!
! C is cool!
11
11
12
11
```

Давайте посмотрим, что происходит. Первый оператор printf() эквивалентен следующему оператору:

```
printf("%s! C is cool!\n","%s! C is cool!\n");
```

Второй оператор вывода сначала увеличивает значение num до 11, а затем выводит значение. Третий оператор вывода выводит значение num, которое равно 11, после чего увеличивает его до 12. Четвертый оператор выводит текущее значение n, которое по-прежнему равно 12, а затем уменьшает n до 11. Заключительный оператор вывода выводит текущее значение переменной num, которое равно 11.

7. Вывод имеет следующий вид:

```
SOS:4 4.00
```

Значение выражения c1 - c2 совпадает со значением выражения 'S' - '0', которое в ASCII-коде выглядит как 83 - 79.

8. Программа выведет одну строку цифр от 1 до 10 в полях шириной по пять символов, а затем перейдет на новую строку:

```
1    2    3    4    5    6    7    8    9   10
```

9. Возможный вариант программы, в которой предполагается, что буквы кодируются последовательно, как это имеет место в кодировке ASCII, выглядит следующим образом:

```
#include <stdio.h>
int main(void)
{
   char ch = 'a';
   while (ch <= 'g')
      printf("%5c", ch++);
   printf("\n");
   return 0;
}
```

10. Эти фрагменты выводили бы следующие результаты:

а.
```
1    2
```

Обратите внимание, что переменная x сначала инкрементируется и затем производится сравнение. Курсор остается в той же строке.

б.
```
101
102
103
104
```

Обратите внимание, что на этот раз x сначала сравнивается, а затем инкрементируется. И в данном случае, и в случае а) значение x увеличивается перед выполнением вывода. Обратите также внимание, что запись второго оператора `printf()` с отступом не делает его частью цикла `while`. Следовательно, этот оператор вызывается только один раз после завершения цикла `while`.

в. `stuvw`

Здесь инкрементирование происходит только после первого оператора `printf()`.

11. Программа сконструирована неудачно. Поскольку оператор `while` не содержит фигурных скобок, частью цикла является только оператор `printf()`, поэтому программа бесконечно повторяет вывод сообщения `COMPUTER BYTES DOG` до тех пор, пока вам не удастся принудительно прекратить ее работу.

12. а. `x = x + 10;`

    б. `x++;` или `++x;` или `x = x + 1;`

    в. `c = 2 * (a + b);`

    г. `c = a + 2* b;`

13. а. `x--;` или `--x;` или `x = x - 1;`

    б. `m = n % k;`

    в. `p = q / (b - a);`

    г. `x = (a + b) / (c * d);`

Ответы на вопросы для самоконтроля из главы 6

1. `2, 7, 70, 64, 8, 2`

2. Он должен вывести следующее:

   `36 18 9 4 2 1`

   Если бы переменная `value` имела тип `double`, то результат проверки условия оставался бы истинным даже при значениях `value`, меньших 1. Выполнение цикла продолжалось бы до тех пор, пока потеря значимости при вычислениях с плавающей запятой не привела бы к получению значения 0. Кроме того, в этом случае выбор спецификатора `%3d` был бы неправильным.

3. а. `x > 5`

   б. `scanf("%lf",&x) != 1`

   в. `x == 5`

4. а. `scanf("%d", &x) == 1`

   б. `x != 5`

   в. `x >= 20`

5. Строка 4: должна содержать `list[10]`.

   Строка 6: запятые необходимо заменить точками с запятой.

   Строка 6: диапазоном для `i` должен быть 0–9, а не 1–10.

Строка 9: запятые необходимо заменить точками с запятой.

Строка 9: операцию >= необходимо заменить операцией <=, иначе цикл будет выполняться порядочное время.

Строка 11: между строками 11 и 12 должна присутствовать дополнительная закрывающая фигурная скобка. Одна скобка закрывает блочный оператор и еще одна – программу. Между этими скобками необходимо поместить строку `return 0;`.

Вот скорректированная версия:

```
#include <stdio.h>
int main(void)
{                                    /* строка 3  */
   int i, j, list[10];               /* строка 4  */

   for (i = 0; i <  10;  i++)        /* строка 6  */
   {                                 /* строка 7  */
      list[i] = 2*i + 3;             /* строка 8  */
      for (j = 1; j <= i; j++)       /* строка 9  */
         printf(" %d", list[j]);     /* строка 10 */
      printf("\n");                  /* строка 11 */
   }
   return 0;
}
```

6. Ниже показан один из возможных вариантов:

```
#include <stdio.h>
int main(void)
{
   int col, row;

   for (row = 1; row <= 4; row++)
   {
      for (col = 1; col <= 8; col++)
         printf("$");
      printf("\n");
   }
   return 0;
}
```

7. а. Программа выведет следующую строку:

   `Hi! Hi! Hi! Bye! Bye! Bye! Bye! Bye!`

   б. Программа выведет следующую строку:

   `ACGM`

   Поскольку код добавляет значение `int` к значению `char`, компилятор может выдать предупреждение о возможности потери значащих цифр.

8. Эти программы выведут следующие данные:

   а. `Go west, youn`

   б. `Hp!xftu-!zpvo`

   в. `Go west, young`

   г. `$o west, youn`

9. Должен быть получен следующий вывод:

```
31|32|33|30|31|32|33|
***
1
5
9
13

***
2 6
4 8
8 10

***
======
=====
====
===
==
```

10. а. mint.

    б. 10 элементов.

    в. Значения типа double.

    г. Правильной является строка scanf("%lf", &mint[2]); mint[2] — это значение типа double, а &mint[2] — его местоположение.

11. Поскольку первый элемент имеет индекс 0, переменная цикла должна изменяться в диапазоне от 0 до SIZE - 1, а не от 1 до SIZE. Однако внесение этого изменения приводит к присваиванию первому элементу значения 0, а не 2. Таким образом, цикл необходимо переписать в следующем виде:

```
for (index = 0; index < SIZE; index++)
   by_twos[index] = 2 * (index + 1);
```

Аналогично должны быть изменены пределы во втором цикле. Кроме того, имя массива должно сопровождаться индексом массива:

```
for( index = 0; index < SIZE; index++)
    printf("%d ", by_twos[index]);
```

Один из опасных аспектов неправильного указания пределов в цикле связан с тем, что программа может работать, но поскольку она помещает данные не в те ячейки памяти, куда должна, она может перестать работать в какой-то момент в будущем, формируя своего рода "мину замедленного действия".

12. Определение должно объявить возвращаемый тип как long и содержать оператор return, который возвращает значение long.

13. Приведение типа num к long гарантирует выполнение вычислений с типом long, а не int. В системе с 16-битным int умножение двух значений типа int порождает результат, который перед возвратом значения усекается до типа int, что может приводить к потере данных.

```
long square(int num)
{
   return ((long) num) * num;
}
```

14. Вывод программы имеет следующий вид:

```
1: Hi!
k = 1
k is 1 in the loop
Now k is 3
k = 3
k is 3 in the loop
Now k is 5
k = 5
k is 5 in the loop
Now k is 7
k = 7
```

Ответы на вопросы для самоконтроля из главы 7

1. Истинным является выражение б).
2. а. `number >= 90 && number < 100`

   б. `ch != 'q' && ch != 'k'`

   в. `(number >= 1 && number <= 9) && number != 5`

   г. Один из возможных вариантов является `!(number >= 1 && number <= 9)`, но выражение `number < 1 || number > 9` проще для понимания.
3. Строка 5: эта строка должна иметь вид `scanf("%d %d", &weight, &height);`. Не забудьте использовать символы `&` в `scanf()`. Кроме того, этой строке должен предшествовать оператор, приглашающий ввести данные.

   Строка 9: в данном случае подразумевается выражение `(height < 72 && height > 64)`. Однако первая часть выражения излишня, поскольку, чтобы программа достигла строки `else if`, значение `height` должно быть меньше 72. Поэтому вполне достаточно использовать выражение `(height > 64)`. Но строка 6 уже гарантирует выполнение этого условия, поэтому никакая дополнительная проверка вообще не требуется и выражение `if else` следует заменить выражением `else`.

   Строка 11: это условие избыточно. Второе подвыражение (`weight` не меньше или равно 300) означает то же, что и первое. В данном случае требуется использовать простое выражение `(weight > 300)`. Однако в этой строке присутствует значительно более серьезная ошибка. Строка 11 связана не с тем оператором `if`! Очевидно, что эта конструкция `else` предназначалась для оператора `if` из строки 6. Однако в соответствии с правилом связывания с ближайшим предшествующим оператором `if` она будет связана с оператором `if` строки 9. Поэтому строка 11 выполняется тогда, когда значение `weight` меньше 100, а значение `height` меньше или равно 64. В результате по достижении этого оператора значение `weight` никак не может превышать 300.

   Строки 7–9: эти строки должны быть заключены в фигурные скобки. Тогда строка 11 станет альтернативой строке 6, а не строке 9. Или же если заменить выражение `if else` в строке 9 выражением `else`, то никакие фигурные скобки не понадобятся.

   Строка 13: ее необходимо упростить до `if (height > 48)`. В действительности эту строку можно вообще удалить, т.к. строка 12 уже делает нужную проверку.

Строка 15: эта конструкция else связана с последним оператором if, указанным в строке 13. Чтобы связать это выражение с оператором if в строке 11, строки 13 и 14 потребуется поместить в фигурные скобки. Или, как было предложено ранее, можно просто избавиться от строки 13.

Вот скорректированная версия:

```
#include <stdio.h>
int main(void)
{
   int weight, height;  /* вес в фунтах, рост в дюймах */

   printf("Введите свой вес в фунтах ");
   printf("и свой рост в дюймах.\n");
   scanf("%d %d", &weight, &height);
   if (weight < 100 && height > 64)
      if (height >= 72)
         printf("Ваш вес слишком мал для вашего роста.\n");
      else
         printf("Ваш вес мал для вашего роста.\n");
   else if (weight > 300 && height < 48)
         printf("Ваш рост мал для вашего веса.\n");
   else
      printf("У вас идеальный вес.\n");

   return 0;
}
```

4. а. 1. Утверждение истинно, и численно это равно 1.
   б. 0. 3 не меньше 2.
   в. 1. Если первое выражение ложно, то второе истинно, и наоборот. Чтобы все выражение было истинным, достаточно истинности только одного из его подвыражений.
   г. 6, поскольку значением 6 > 2 является 1.
   д. 10, т.к. проверяемое условие истинно.
   е. 0. Если выражение x > y истинно, то значением выражения будет y > x, которое в этом случае ложно, или равно 0. Если выражение x > y ложно, значением выражения будет x > y, которое в данном случае ложно.
5. Программа выведет следующую строку:

```
*#%*#%$#%*#%*#%$#%*#%*#%$#%*#%*#%
```

   Несмотря на присутствующие в коде отступы, символ # выводится на каждой итерации цикла, т.к. этот оператор вывода не является частью составного оператора.
6. Программа выводит следующие данные:

```
fat hat cat Oh no!
hat cat Oh no!
cat Oh no!
```

7. Комментарии в строках 5–7 должны завершаться символами */ либо же символы /* можно заменить символами //. Выражение 'a' <= ch >= 'z' потребуется заменить следующим выражением:

```
ch >= 'a' && ch <= 'z'
```

В качестве альтернативы можно воспользоваться более простым и переносимым подходом, включив файл ctype.h и вызвав функцию islower(). Кстати, выражение 'a' <= ch >= 'z' с точки зрения синтаксиса С допустимо; оно лишь затрудняет понимание его смысла. Поскольку операции отношения ассоциируются слева направо, это выражение интерпретируется как ('a' <= ch) >= 'z'. Выражение в скобках принимает значение 1 или 0 (истинно или ложно), и это значение проверяется на предмет того, больше оно или равно числовому коду 'z'. Ни 0, ни 1 не удовлетворяют этому условию, поэтому значение всего выражения всегда равно 0 (ложно). Во втором условном выражении символы || необходимо заменить символами &&. Кроме того, хотя выражение !(ch < 'A') является допустимым и правильным по смыслу, выражение !(ch < 'A') проще. За выражением 'Z' должны следовать две закрывающие скобки, а не одна. Здесь снова проще воспользоваться функцией isupper(). Оператору oc++; должна предшествовать конструкция else. В противном случае он будет инкрементировать каждый символ. Управляющее выражение в printf() должно быть заключено в двойные кавычки.

Ниже показана скорректированная версия:

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   char ch;
   int lc = 0;    /* счетчик строчных символов  */
   int uc = 0;    /* счетчик прописных символов */
   int oc = 0;    /* счетчик других символов    */

   while ((ch = getchar()) != '#')
   {
      if (islower(ch))
         lc++;
      else if (isupper(ch))
         uc++;
      else
         oc++;
   }
   printf("%d строчных, %d прописных, %d других", lc, uc, oc);

   return 0;
}
```

8. К сожалению, она бесконечно выводит одну и ту же строку:

```
Вам 65. Получите свои золотые часы.
```

Проблема в том, что строка

```
if (age = 65)
```

устанавливает переменную age в 65, что на каждой итерации цикла интерпретируется как истинное.

9. Результат выполнения программы с введенными значениями выглядит следующим образом:

```
q
Шаг 1
Шаг 2
```

```
Шаг 3
c
Шаг 1
h
Шаг 1
Шаг 3
b
Шаг 1
Готово
```

Обратите внимание, что ввод b и # прекращает цикл, но ввод b обеспечивает вывод строки Шаг 1, а ввод # — нет.

10. Вот одно из возможных решений:

```
#include <stdio.h>
int main(void)
{
    char ch;
    while ((ch = getchar()) != '#')
    {
        if (ch != '\n')
        {
            printf("Шаг 1\n");
            if (ch == 'b')
                break;
            else if (ch != 'c')
            {
                if (ch != 'h')
                    printf("Шаг 2\n");
                printf("Шаг 3\n");
            }
        }
    }
    printf("Готово\n");
    return 0;
}
```

Ответы на вопросы для самоконтроля из главы 8

1. Выражение putchar(getchar()) вынуждает программу прочитать следующий входной символ и вывести его; возвращаемое значение getchar() служит аргументом putcar(). Нет, выражение getchar(putchar()) не является допустимым, поскольку функция getchar() не принимает аргумент, а функции putcar() он необходим.

2. а. Отображает символ H.

   б. Если система использует кодировку ASCII, то этот оператор вызывает подачу звукового сигнала.

   в. Перемещает курсор в начало следующей строки.

   г. Вызывает затирание последнего выведенного символа и возврат курсора на один символ назад.

3. count <essay >essayct или по-другому count >essayct <essay

4. Допустимых вариантов нет.

5. Это сигнал (специальное значение), возвращаемый функциями getchar() и scanf() для указания на то, что они обнаружили конец файла.

6. а. Вывод имеет следующий вид:

   ```
   If you qu
   ```

   Обратите внимание, что символ I отличается от символа i. Кроме того, символ i не выводится, т.к. выход из цикла происходит до обнаружения этого символа.

   б. Если в системе применяется кодировка ASCII, то вывод будет таким:

   ```
   HJacrthjacrt
   ```

   При первом проходе значением переменной ch является символ H. Операция ch++ приводит к использованию (выводу) этого значения, а затем к его инкрементированию (до I). После этого операция ++ch инкрементирует значение (до J) и применяет его (выводит). Затем читается следующий символ (a) и процесс повторяется. Важно отметить, что операции инкремента воздействуют на значение ch после присваивания значения этой переменной; они не вызывают какое-то перемещение во входной очереди.

7. Стандартная библиотека ввода-вывода C отображает различные формы файлов на унифицированные потоки, которые могут обрабатываться одинаково.

8. Числовой ввод пропускает символы пробела и новой строки, а символьный ввод — нет. Предположим, что имеется следующий код:

   ```
   int score;
   char grade;
   printf("Введите количество баллов.\n");
   scanf("%s", &score);
   printf("Введите букву квалификации.\n");
   grade = getchar();
   ```

   Если в качестве количества баллов (score) ввести число 98 и нажать клавишу <Enter> для передачи этого значения программе, то ей также будет передан символ новой строки, который станет следующим входным символом и прочитается в grade как значение рейтинга. Если числовой ввод предшествует символьному, то в программе должен быть предусмотрен код, предназначенный для отбрасывания символа новой строки перед выполнением символьного ввода.

Ответы на вопросы для самоконтроля из главы 9

1. Формальный параметр — это переменная, которая определена в вызываемой функции. Фактический аргумент — это значение, присутствующее в вызове функции; это значение присваивается формальному параметру. Фактический аргумент можно считать значением, которым инициализируется формальный параметр при вызове функции.

2. а. void donut(int n)

   б. int gear(int t1, int t2)

   в. int guess(void)

   г. void stuff_it(double d, double *pd)

3. а. `char n_to_char(int n)`

   б. `int digits(double x, int n)`

   в. `double * which(double * p1, double * p2)`

   г. `int random(void)`

4.
```
int  sum(int  a,  int  b)
{
   return a + b;
}
```

5. Необходимо заменить все вхождения типа `int` типом `double`:
```
double sum(double a, double b)
{
   return a + b;
}
```

6. В этой функции необходимо использовать указатели:
```
void alter(int * pa, int * pb)
{
   int temp;
   temp = *pa + *pb;
   *pb = *pa - *pb;
   *pa = temp;
}
```
   или
```
void alter(int * pa, int * pb)
{
   *pa += *pb;
   *pb = *pa - 2 * *pb;
}
```

7. Да, ошибки присутствуют. Аргумент `num` должен быть объявлен в списке аргументов функции `salami()`, а не после фигурной скобки. Кроме того, вместо `num++` должно быть `count++`.

8. Ниже показано одно возможное решение:
```
int largest(int a, int b, int c)
{
   int max = a;
   if (b > max)
      max = b;
   if (c > max)
      max = c;
   return max;
}
```

9. Минимальная по объему кода программа приведена ниже. Функции `showmenu()` и `getchoice()` являются возможными решениями для пунктов а) и б).
```
#include <stdio.h>
void showmenu(void);    /* объявление используемых функций */
int getchoice(int, int);
int main()
{
   int res;
```

```
    showmenu();
    while ((res = getchoice(1,4)) != 4)
    {
       printf("Меня устраивает вариант %d.\n", res);
       showmenu();
    }
    printf("Программа завершена.\n");
    return 0;
}

void showmenu(void)
{
    printf("Выберите один из следующих вариантов:\n");
    printf("1) копировать файлы     2) переместить файлы\n");
    printf("3) удалить файлы        4) выйти из программы\n");
    printf("Введите номер выбранного варианта:\n");
}

int getchoice(int low, int high)
{
    int ans;
    int good;
    good = scanf("%d", &ans);
    while (good == 1 && (ans < low || ans  > high))
    {
      printf("%d является недопустимым вариантом; повторите попытку\n", ans);
      showmenu();
      scanf("%d", &ans);
    }
    if (good != 1)
    {
       printf("Нечисловой ввод.");
       ans = 4;
    }
    return ans;
}
```

Ответы на вопросы для самоконтроля из главы 10

1. Вывод выглядит следующим образом:

```
8 8
4 4
0 0
2 2
```

2. Массив `ref` содержит четыре элемента, поскольку таково количество значений в списке инициализации.

3. Имя массива `ref` указывает на первый элемент массива — целое число `8`. Выражение `ref + 1` указывает на второй элемент — целое число `4`. Конструкция `++ref` не является допустимым выражением C; `ref` представляет собой константу, а не переменную.

4. ptr указывает на первый элемент, а ptr + 2 — на третий элемент, который будет первым элементом второй строки.

   а. 12 и 16.

   б. 12 и 14 (согласно скобкам, в первую строку попадает только число 12).

5. ptr указывает на первую строку, а ptr+1 — на вторую строку; *ptr указывает на первый элемент в первой строке, а *(ptr + 1) — на первый элемент второй строки.

   а. 12 и 16.

   б. 12 и 14 (согласно скобкам, в первую строку попадает только число 12).

6. а. &grid[22][56]

   б. &grid[22][0] или grid[22]

   (Второй вариант представляет собой имя одномерного массива, состоящего из 100 элементов, т.е. адрес его первого элемента, которым является grid[22][0].)

   в. &grid[0][0] или grid[0] или (int *) grid.

   (Здесь grid[0] — это адрес элемента grid[0][0] типа int, а grid — адрес 100-элементного массива grid[0]. Упомянутые два адреса имеют одно и то же значение, но разные типы; приведение делает типы одинаковыми.)

7. а. int digits[10];

   б. float rates[6];

   в. int mat[3][5];

   г. char * psa[20];

   Обратите внимание, что приоритет [] выше приоритета *, поэтому при отсутствии скобок сначала применяется описатель массива, а затем описатель указателя. Таким образом, это объявление эквивалентно объявлению char * (psa[20]);.

   д. char (*pstr)[20];

На заметку!

В пункте д) нельзя использовать объявление char *pstr[20];. Это сделало бы pstr массивом указателей, а не указателем на массив. В частности, pstr указывал бы на одиночное значение char — первый элемент массива, а pstr + 1 указывал бы на следующий байт. При корректном объявлении pstr представляет собой переменную, а не имя массива, и pstr + 1 указывает на позицию, которая на 20 байт отстоит от начального байта.

8. а. int sextet[6] = {1, 2, 4, 8, 16, 32};

   б. sextet[2]

   в. int lots[100] = { [99] = -1};

   г. int pots[100] = { [5] = 101, [10] = 101, 101, 101, 101};

9. От 0 до 9.

10. а. rootbeer[2] = value;

    Допустим.

    б. scanf("%f", &rootbeer);

    Недопустим; &rootbeer не является значением типа float.

в. `rootbeer = value;`

Недопустим; `rootbeer` не является значением типа `float`.

г. `printf("%f", rootbeer);`

Недопустим; `rootbeer` не является значением типа `float`.

д. `things[4][4] = rootbeer[3];`

Допустим.

е. `things[5] = rootbeer;`

Недопустим; нельзя присваивать массивы.

ж. `pf = value;`

Недопустим; `value` не является адресом.

з. `pf = rootbeer;`

Допустим.

11. `int screen[800][600];`

12. а.
```
void process(double ar[], int n);
void processvla(int n, double ar[n]);
process(trots, 20);
processvla(20, trots);
```
б.
```
void process2(short ar2[30], int n);
void process2vla(int n, int m, short ar2[n][m]);
process2(clops, 10);
process2vla(10, 30, clops);
```
в.
```
void process3(long ar3[10][15], int n);
void process3vla(int n, int m,int k, long ar3[n][m][k]);
process3(shots, 5);
process3vla(5, 10, 15, shots);
```

13. а. `show( (int [4]) {8,3,9,2}, 4);`

б. `show2( (int [][3]){{8,3,9}, {5,4,1}}, 2);`

Ответы на вопросы для самоконтроля из главы 11

1. Если вы хотите, чтобы результат был строкой, то инициализация должна включать `'\0'`. Разумеется, альтернативный синтаксис добавляет нулевой символ автоматически:

```
char name[] = "Fess";
```

2.
```
Увидимся завтра в кафе.
видимся завтра в кафе.
Увидимс
идимс
```

3.
```
о
но
сно
усно
кусно
Вкусно
```

4. `За всю дорогу я смог осилить лишь часть.`

5. а. `Хо Хо Хо!!оХ оХ оХ`

   б. Указатель на `char` (т.е. `char *`)

   в. Адрес начальной буквы `Х`.

   г. Выражение `*--pc` означает уменьшение указателя на 1 и использование значения, находящегося по этому адресу. `--*pc` означает взятие значения, на которое ссылается указатель `pc`, и уменьшение этого значения на 1 (например, символ `Х` становится символом `Ф`).

   д. `Хо Хо Хо!!оХ оХ о`

На заметку!

Между символами `!` и `!` присутствует нулевой символ, но обычно он не оказывает никакого влияния на вывод.

   е. `while(*pc)` проверяет, не указывает ли `pc` на нулевой символ (т.е. на конец строки). В выражении используется значение, расположенное по указанному месту.

   `while(pc - str)` проверяет, не указывает ли `pc` на то же место, что и `str` (начало строки). В выражении применяются значения самих указателей.

   ж. После первой итерации цикла `while` указатель `pc` указывает на нулевой символ. При входе во вторую итерацию цикла он указывает на ячейку памяти, предшествующую нулевому символу (т.е. расположенную непосредственно перед той, на которую указывает `str`). Этот байт интерпретируется как символ и выводится. Затем указатель возвращается к предыдущему байту. Условие выхода из цикла `(pc == str)` никогда не удовлетворяется, и процесс продолжается до тех пор, пока не будет прерван пользователем или системой.

   з. `pr()` должен быть объявлен в вызывающей программе:

   ```
   char * pr(char *);
   ```

6. Под символьные переменные отводится один байт, поэтому `sign` занимает один байт. Но символьная константа сохраняется в виде `int`, т.е. `'$'` обычно будет использовать 2 или 4 байта; тем не менее, для хранения кода `'$'` в действительности будет задействован только один байт из `int`. Строка `"$"` использует два байта: один для хранения кода символа `'$'` и еще один для хранения кода символа `'\0'`.

7. Эта программа выводит следующие данные:

   ```
   How are ya, sweetie? How are ya, sweetie?
   Beat the clock.
   eat the clock.
   Beat the clock. Win a toy.
   Beat
   chat
   hat
   at
   ```

```
t
t
at
How are ya, sweetie?
```

8. Ее вывод имеет следующий вид:

```
faavrhee
*le*on*sm
```

9. Ниже показано одно из возможных решений:

```
#include <stdio.h>     // для fgets(), getchar()
char * s_gets(char * st, int n)
{
   char * ret_val;
   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      while (*st != '\n' && *st != '\0')
         st++;
      if (*st == '\n')
         *st = '\0';
      else
         while (getchar() != '\n')
            continue;
   }
   return ret_val;
}
```

10. Вот одно из возможных решений:

```
int strlen(const char * s)
{
   int ct = 0;
   while (*s++)          // или while (*s++ != '\0')
      ct++;
   return(ct);
}
```

11. Ниже показано одно из возможных решений:

```
#include <stdio.h>      // для fgets(), getchar()
#include <string.h>     // для strchr();
char * s_gets(char * st, int n)
{
   char * ret_val;
   char * find;
   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      find = strchr(st, '\n');   // поиск символа новой строки
      if (find)                  // если адрес не является NULL,
         *find = '\0';           // поместить туда нулевой символ
      else
         while (getchar() != '\n')
            continue;
   }
   return ret_val;
}
```

12. Одно из возможных решений выглядит так:

```
#include <stdio.h>      /* для определения NULL                          */
char * strblk(char * string)
{
   while (*string != ' ' && *string != '\0')
      string++;        /* остановиться на первом пробеле или нулевом
символе */
   if (*string == '\0')
      return NULL;     /* NULL - это нулевой указатель                    */
   else
      return string;
}
```

Второе решение предотвращает изменение строки функцией, но позволяет применять возвращаемое значение для изменения строки. Выражение (char *) string называют "избавлением от const".

```
#include <stdio.h>  /* для определения NULL                              */
char * strblk(const char * string)
{
   while (*string != ' ' && *string != '\0')
      string++;    /* остановиться на первом пробеле или нулевом символе */
   if (*string == '\0')
      return NULL; /* NULL - это нулевой указатель                        */
   else
      return (char *) string;
}
```

13. Ниже показано возможное решение:

```
/* compare.c -- это будет работать */
#include <stdio.h>
#include <string.h>   // объявление strcmp()
#include <ctype.h>
#define ANSWER "GRANT"
#define SIZE 40
char * s_gets(char * st, int n);
void ToUpper(char * str);

int main(void)
{
   char try[SIZE];

   puts("Кто похоронен в могиле Гранта?");
   s_gets(try, SIZE);
   ToUpper(try);
   while (strcmp(try,ANSWER) != 0)
   {
      puts("Неправильно! Попытайтесь еще раз.");
      s_gets(try, SIZE);
      ToUpper(try);
   }
   puts("Теперь правильно!");

   return 0;
}
```

```
void ToUpper(char * str)
{
   while (*str != '\0')
   {
      *str = toupper(*str);
      str++;
   }
}
char * s_gets(char * st, int n)
{
   char * ret_val;
   int i = 0;
   ret_val = fgets(st, n, stdin);
   if (ret_val)
   {
      while (st[i] != '\n' && st[i] != '\0')
         i++;
      if (st[i] == '\n')
         st[i] = '\0';
      else // требуется наличие words[i] == '\0'
         while (getchar() != '\n')
            continue;
   }
   return ret_val;
}
```

Ответы на вопросы для самоконтроля из главы 12

1. Автоматический класс хранения, регистровый класс хранения и статический класс хранения без связывания.
2. Статический класс хранения без связывания, статический класс хранения с внутренним связыванием и статический класс хранения с внешним связыванием.
3. Статический класс хранения с внешним связыванием. Статический класс хранения с внутренним связыванием.
4. Они не имеют связывания.
5. Ключевое слово extern используется в объявлениях для указания переменной или функции, которая объявлена в каком-то другом месте.
6. Оба оператора выделяют память под массив из 100 значений int. Оператор, в котором используется calloc(), дополнительно устанавливает каждый элемент в 0.
7. Переменная daisy известна функции main() по умолчанию, а функциям petal(), stem() и root() — благодаря объявлению extern. Объявление extern int daisy; во втором файле делает переменную daisy известной всем функциям в этом файле. Первая переменная lily является локальной для функции main(). Ссылка на переменную lily в функции petal() — ошибка, поскольку ни один из файлов не содержит объявления внешней переменной lily. Существует внешняя статическая переменная lily, но она известна только функциям из второго файла. Первая внешняя переменная rose известна функции root(), но функция stem() заменяет ее собственной локальной переменной rose.

8. Вывод будет следующим:

```
color в main() равно B
color в first() равно R
color в main() равно B
color в second() равно G
color в main() равно G
```

Функция first() не использует глобальную переменную color, но ее использует функция second().

9. а. Они говорят о том, что программа будет использовать переменную plink, которая локальна для файла, содержащего функцию. Первый аргумент функции value_ct() — это указатель на целочисленное значение, которое, по всей видимости, является первым элементом массива, состоящего из n членов. В данном случае важно отметить, что программа не сможет применять указатель arr для изменения значений в исходном массиве.

   б. Нет. Аргументы value и n уже являются копиями исходных данных, поэтому функция никак не может изменять соответствующие значения в вызывающей программе. Эти объявления предотвращают изменение значений value и n внутри самой функции. Например, функция не могла бы использовать выражение n++, если бы объявление n было снабжено const.

Ответы на вопросы для самоконтроля из главы 13

1. Программа должна содержать строку #include <stdio.h>, чтобы можно было использовать определения этого файла. Переменная fp должна быть объявлена как файловый указатель: FILE *fp;. Функция fopen() требует указания режима: fopen("gelatin", "w") или, возможно режима "a". Порядок следования аргументов функции fputs() должен быть обратным. Для повышения удобочитаемости строка вывода должна иметь символ новой строки, поскольку fputs() не добавляет его автоматически. Функция fclose() требует передачи в качестве аргумента файлового указателя, а не имени файла: fclose(fp);. Ниже показана скорректированная версия:

```
#include <stdio.h>
int main(void)
{
    FILE * fp;
    int k;

    fp = fopen("gelatin", "w");
    for (k = 0; k < 30; k++)
        fputs("Кто-то ест студень.\n", fp);
    fclose(fp);

    return 0;
}
```

2. По возможности она будет открывать файл, имя которого задано в первом аргументе командной строки, и выводить на экран каждый присутствующий в файле цифровой символ.

3. а. `ch = getc(fp1);`

   б. `fprintf(fp2,"%c"\n",ch);`

   в. `putc(ch,fp2);`

   г. `fclose(fp1); /* закрыть файл terky */`

На заметку!

Указатель `fp1` используется для операций ввода, поскольку он идентифицирует файл, открытый в режиме чтения. Подобным же образом файл, на который указывает `fp2`, был открыт в режиме записи, поэтому он применяется с функциями вывода.

4. Ниже демонстрируется один из подходов:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char * argv[])
{
   FILE * fp;
   double n;
   double sum = 0.0;
   int ct = 0;

   if (argc == 1)
      fp = stdin;
   else if (argc == 2)
   {
      if ((fp = fopen(argv[1], "r")) == NULL)
      {
         fprintf(stderr, "Не удается открыть %s\n", argv[1]);
         exit(EXIT_FAILURE);
      }
   }
   else
   {
      fprintf(stderr, "Использование: %s [имя_файла]\n", argv[0]);
      exit(EXIT_FAILURE);
   }
   while (fscanf(fp, "%lf", &n) == 1)
   {
      sum += n;
      ++ct;
   }
   if (ct > 0)
      printf("Среднее арифметическое %d значений = %f\n", ct, sum / ct);
   else
      printf("Допустимые данные отсутствуют.\n");

   return 0;
}
```

5. Одно из возможных решений выглядит так:

```
#include <stdio.h>
#include <stdlib.h>
#define BUF 256
int has_ch(char ch, const char * line);
```

```
int main(int argc,char * argv[])
{
   FILE * fp;
   char ch;
   char line [BUF];
   if (argc != 3)
   {
      printf("Использование: %s символ имя_файла\n", argv[0]);
      exit(EXIT_FAILURE);
   }
   ch = argv[1][0];
   if ((fp = fopen(argv[2], "r")) == NULL)
   {
      printf("Не удается открыть %s\n", argv[2]);
      exit(EXIT_FAILURE);
   }
   while (fgets(line,BUF,fp) != NULL)
   {
      if (has_ch(ch,line))
         fputs(line,stdout);
   }
   fclose(fp);
   return 0;
}
int has_ch(char ch, const char * line)
{
   while (*line)
      if (ch == *line++)
         return(1);
   return 0;
}
```

Функции `fgets()` и `fputs()` работают вместе, поскольку `fgets()` оставляет в строке символ новой строки `\n`, созданный в результате нажатия клавиши <Enter>, а функция `fputs()` не добавляет его, как это делает `puts()`.

6. Отличие между двоичным и текстовым файлами определяется системно-зависимыми особенностями этих файловых форматов. Отличие между двоичным и текстовым потоками связано с преобразованиями, выполняемыми программой во время чтения или записи потоков. (В двоичном потоке преобразования не делаются; в текстовом потоке могут осуществляться преобразования символов новой строки и других символов.)

7. а. Сохранение числа 8238201 с помощью функции `fprintf()` приводит к его сохранению в виде семи символов, занимающих 7 байтов. При использовании функции `fwrite()` число сохраняется в виде двоичного представления 4-байтового целого числового значения.

   б. Ни в чем. В обоих случаях символ сохраняется в виде 1-байтового двоичного кода.

8. Первый оператор представляет собой всего лишь сокращенную форму записи второго оператора. Третий оператор выполняет запись в стандартный вывод ошибок. Обычно стандартные сообщения об ошибках направляются туда же, куда и стандартный вывод, но переадресация стандартного вывода не оказывает влияния на стандартный вывод ошибок.

9. Режим "r+" позволяет выполнять чтение и запись в любом месте файла, поэтому он наиболее подходит в данном случае. Режим "a+" позволяет только дописывать данные в конец файла, а режим "w+" начинает работу с “чистого листа”, удаляя предыдущее содержимое файла.

Ответы на вопросы для самоконтроля из главы 14

1. Правильным ключевым словом является `struct`, а не `structure`. Шаблон требует наличия либо дескриптора перед открывающей скобкой, либо имени переменной после закрывающей скобки. Кроме того, символ точки с запятой должен присутствовать после выражения `* togs` и в конце шаблона.

2. Вывод имеет следующий вид:

```
6 1
22 Spiffo Road
S p
```

3.
```
struct month {
    char name[10];
    char abbrev[4];
    int days;
    int monumb;
};
```

4.
```
struct month months[12] =
{
    {"Январь", "янв", 31, 1},
    {"Февраль", "фев", 28, 2},
    {"Март", "мар", 31, 3},
    {"Апрель", "апр", 30, 4},
    {"Май", "май", 31, 5},
    {"Июнь", "июн", 30, 6},
    {"Июль", "июл", 31, 7},
    {"Август", "авг", 31, 8},
    {"Сентябрь", "сен", 30, 9},
    {"Октябрь", "окт", 31, 10},
    {"Ноябрь", "ноя", 30, 11},
    {"Декабрь", "дек", 31, 12}
};
```

5.
```
extern struct month months[];
int days(int month)
{
    int index, total;
    if (month < 1 || month > 12)
        return(-1); /* сигнал ошибки */
    else
    {
        for (index = 0, total = 0; index < month; index ++)
            total += months[index].days;
        return( total);
    }
}
```

Обратите внимание, что значение index на единицу меньше номера месяца. Это связано с тем, что нумерация индексов в массивах начинается с 0. Поэтому необходимо использовать условие проверки index < month, а не index <= month.

6. а. Чтобы можно было использовать функцию strcpy(), включите заголовочный файл string.h:

```
typedef struct lens { /* дескриптор структуры lens          */
   float foclen;      /* фокусное расстояние в миллиметрах  */
   float fstop;       /* диафрагма                          */
   char brand[30];    /* марка производителя                */
} LENS;

LENS bigEye[10];
bigEye[2].foclen = 500;
bigEye[2].fstop = 2.0;
strcpy(bigEye[2].brand, "Remarkatar");
```

   б. `LENS bigEye[10] = { [2] = {500, 2, "Remarkatar"} };`

7. а. 6

   Arcturan

   cturan

   б. Для этого можно использовать имя структуры и указатель:

```
deb.title.last
pb->title.last
```

   в. Одна из возможных версий выглядит так:

```
#include <stdio.h>
#include "starfolk.h" /* обеспечение доступности определений структуры*/
void prbem (const struct bem * pbem)
{
   printf("%s %s - это %d-конечный %s.\n", pbem->title.first,
          pbem->title.last, pbem->limbs, pbem->type);
}
```

8. а. willie.born

   б. pt->born

   в. scanf("%d", &willie.born);

   г. scanf("%d", &pt->born);

   д. scanf("%s", willie.name.lname);

   е. scanf("%s", pt->name.lname);

   ж. willie.name.fname[2]

   з. strlen(willie.name.fname) + strlen(willie.name.lname)

9. Один из возможных вариантов имеет вид:

```
struct car {
   char name[20];
   float hp;
   float epampg;
   float wbase;
   int year;
};
```

10. Функции можно было бы реализовать следующим образом:

```
struct gas {
   float distance;
   float gals;
   float mpg;
};

struct gas mpgs(struct gas trip)
{
   if (trip.gals > 0)
      trip.mpg = trip.distance / trip.gals ;
   else
      trip.mpg = -1.0;
   return trip;
}
void set_mpgs(struct gas * ptrip)
{
   if (ptrip->gals > 0)
      ptrip->mpg = ptrip->distance / ptrip->gals ;
   else
      ptrip->mpg = -1.0;
}
```

Обратите внимание, что первая функция не может напрямую изменять значения из вызывающей программы, поэтому для передачи информации необходимо использовать возвращаемое значение:

```
struct gas idaho = {430.0, 14.8}; // установка значений двух первых членов
idaho = mpgs(idaho);              // переустановка структуры
```

Однако вторая функция обращается к исходной структуре непосредственно:

```
struct gas ohio = {583, 17.6};    // установка значений двух первых членов
set_mpgs(ohio);                   // установка значения третьего члена
```

11. `enum choices {no, yes, maybe};`

12. `char * (*pfun)(char *, char);`

13.
```
double sum(double, double);
double diff(double, double);
double times(double, double);
double divide(double, double);
double (*pf1[4])(double, double) = {sum, diff, times, divide};
```

Чтобы упростить код, последнюю строку можно заменить следующими двумя строками:

```
typedef double (*ptype)(double, double);
ptype pf[4] = {sum, diff, times, divide};
```

Функция `diff()` вызывается следующим образом:

```
pf1[1](10.0, 2.5);     // первая форма записи
(*pf1[1])(10.0, 2.5);  // эквивалентная форма записи
```

Ответы на вопросы для самоконтроля из главы 15

1. а. 00000011
   б. 00001101
   в. 00111011
   г. 01110111
2. а. 21, 025, 0x15
   б. 85, 0125, 0x55
   в. 76, 0114, 0x4C
   г. 157, 0235, 0x9D
3. а. 252
   б. 2
   в. 7
   г. 7
   д. 5
   е. 3
   ж. 28
4. а. 255
   б. 1 ("не ложно" – это "истинно")
   в. 0
   г. 1 ("истинно" И "истинно" – это "истинно")
   д. 6
   е. 1 ("истинно" ИЛИ "истинно" – это "истинно")
   ж. 40
5. В двоичной форме маска имеет вид 1111111, в десятичной – 127, в восьмеричной 0177, а в шестнадцатеричной – 0x7F.
6. Оба выражения bitval *= 2 и bitval << 1 удваивают текущее значение переменной bitval, поэтому они эквивалентны. Однако выражения mask += bitval и mask |= bitval оказывают одинаковое влияние, только если переменные bitval и mask не имеют ни одного общего установленного бита. Например, 2 | 4 равно 6, но этому же значению равен результат выражения 3 | 6.
7. а.
```
struct tb_drives {
    nsigned int diskdrives  : 2;
    unsigned int            : 1;
    unsigned int cdromdrives: 2;
    unsigned int            : 1;
    unsigned int harddrives : 2;
};
```
   б.
```
struct kb_drives {
    unsigned int harddrives : 2;
    unsigned int            : 1;
    unsigned int cdromdrives: 2;
    unsigned int            : 1;
    unsigned int diskdrives : 2;
};
```

Ответы на вопросы для самоконтроля из главы 16

1. а. Результирующий код `dist = 5280 * miles;` является допустимым.

   б. Результирующий код `plort = 4 * 4 + 4;` является допустимым, но если в действительности пользователю необходимо `4 * (4 + 4)`, то должно использоваться `#define POD (FEET + FEET)`.

   в. Результирующий код `nex = = 6;;` является недопустимым. (Если между двумя знаками равенства пробелы отсутствуют, код будет допустимым, но бесполезным.) Очевидно, пользователь забыл, что пишет макрос для препроцессора, а не код на С.

   г. Результирующий код `y = y + 5` является допустимым. Код `berg = berg + 5 * lob;` также допустим, но, скорее всего, представляет собой не тот результат, который хотел получить пользователь. Код `est = berg + 5/y + 5;` допустим, но, вероятно, представляет собой не тот результат, который хотел получить пользователь. Код `nilp = lob *-berg + 5;` является допустимым, но, скорее всего, представляет не тот результат, к которому стремился пользователь.

2. `#define NEW(X) ((X) + 5)`

3. `#define MIN(X,Y) ( (X) < (Y) ? (X) : (Y) )`

4. `#define EVEN_GT(X,Y) ( (X) > (Y) && (X) % 2 == 0 ? 1 : 0 )`

5. `#define PR(X,Y) printf(#X " = %d и " #Y " = %d\n", X,Y)`

   Поскольку в этом макросе `X` и `Y` не появляются в каких-то других операциях (таких как умножение), круглые скобки можно не применять.

6. а. `#define QUARTERCENTURY 25`

   б. `#define SPACE ' '`

   в. `#define PS() putchar(' ')`

   или

   `#define PS() putchar(SPACE)`

   г. `#define BIG(X) ((X) + 3)`

   д. `#define SUMSQ(X,Y) ((X)*(X) + (Y)*(Y))`

7. Попробуйте воспользоваться следующим определением:

   ```
   #define P(X) printf("имя: "#X";  значение: %d;  адрес: %p\n", X, &X)
   ```

   Если используемая вами реализация не распознает спецификатор адреса `%p`, применяйте `%u` или `%lu`.

8. Используйте директивы условной компиляции. Один из возможных способов предусматривает применение директивы `#ifdef`:

   ```
   #define _SKIP_ /* удалите эту строку, если не хотите пропускать код */
   #ifndef _SKIP_
      /* код, который нужно пропустить */
   #endif
   ```

9. ```
   #ifdef PR_DATE
      printf("Дата = %s\n", __DATE__);
   #endif
   ```
   ```

10. Первая версия возвращает значение x*x. При этом просто возвращается значение типа double квадрата x. Например, square(1.3) возвратит 1.69. Вторая версия возвращает (int)(x*x). Здесь результат усекается до типа int. Из-за того, что возвращаемым типом является double, значение int затем повышается до double. Таким образом, 1.69 сначала преобразуется в 1, после чего – в 1.00. Третья версия возвращает (int)(x*x+0.5). Добавление 0.5 приводит к округлению до ближайшего целого числа вместо его усечения. Следовательно, 1.69 становится 2.19, которое усекается до 2 и затем преобразуется в 2.00. Но 1.44 становится 1.94, что усекается до 1 и после этого преобразуется в 1.00.

11. Вот одна из возможных версий:

```
#define BOOL(X)  _Generic((X), _Bool : "boolean", default : "not boolean")
```

12. Аргумент argv должен быть объявлен с типом char *argv[]. Аргументы командной строки хранятся в виде строк, поэтому сначала программа должна преобразовать строку, хранящуюся в элементе массива argv[1], в значение типа double – например, с помощью функции atof() из библиотеки stdlib.h. Чтобы можно было использовать функцию sqrt(), в программу потребуется включить заголовочный файл math.h. Прежде чем извлекать квадратный корень, программа должна выполнить проверку на предмет передачи отрицательных значений.

13. а. Вызов функции должен выглядеть следующим образом:

```
qsort( (void *)scores, (size_t) 1000, sizeof (double), comp);
```

б. Ниже показано подходящее определение функции сравнения:

```
int comp(const void * p1, const void * p2)
{
   /* Для получения доступа к значениям необходимо   */
   /* использовать указатели на константу int        */
   /* Приведения типов не обязательны в C,           */
   /* но обязательны в C++                           */
   const int * a1 = (const int *) p1;
   const int * a2 = (const int *) p2;

   if (*a1 > *a2)
      return -1;
   else if (*a1 == *a2)
      return 0;
   else
      return 1;
}
```

14. а. Вызов функции должен выглядеть примерно так:

```
memcpy(data1, data2, 100 * sizeof(double));
```

б. Вызов функции должен выглядеть следующим образом:

```
memcpy(data1, data2 + 200 , 100 * sizeof(double));
```

Ответы на вопросы для самоконтроля из главы 17

1. Определение типа данных заключается в определении способа хранения данных и набора функций манипулирования данными.

2. Обход списка может выполняться только в одном направлении, поскольку каждая структура содержит адрес следующей, но не предыдущей структуры. Определение структуры можно было бы изменить, чтобы каждая структура содержала два указателя — один на предыдущую структуру и один на следующую. Разумеется, программа должна была бы присваивать соответствующие адреса этим указателям при каждом добавлении новой структуры.

3. ADT — аббревиатура от abstract data type (абстрактный тип данных). ADT представляет собой формальное определение свойств типа и операций, которые можно выполнять с этим типом. ADT должен быть выражен в обобщенных терминах, а не терминах какого-то конкретного языка программирования или деталей реализации.

4. Преимущества передачи переменной напрямую. Данная функция проверяет очередь, но не должна ее изменять. Передача переменной, представляющей очередь, напрямую означает, что функция работает с копией исходных данных, гарантируя невозможность их изменения функцией. При передаче переменной напрямую не нужно помнить о необходимости использования операции взятия адреса или указателя.

   Недостатки передачи переменной напрямую. Программа должна зарезервировать достаточный объем памяти для хранения переменной, а затем скопировать информацию из оригинала в копию. Если переменная представляет собой крупную структуру, ее использование будет сопряжено с большими затратами времени и памяти.

   Преимущества передачи адреса переменной. Если переменная является крупной структурой, то передача адреса и доступ к исходным данным выполняются быстрее и требуют меньшего объема памяти, чем при передаче переменной напрямую.

   Недостатки передачи адреса переменной. Необходимо помнить о применении операции взятия адреса или указателя. В K&R C функция могла бы неумышленно изменить исходные данные, но этой опасности можно избежать, используя спецификатор `const` стандарта ANSI C.

5. a.

Имя типа:	Стек.
Свойства типа:	Может содержать упорядоченную последовательность элементов.
Операции типа:	Инициализация стека пустым значением.
	Определение, является ли стек пустым.
	Определение, является ли стек полным.
	Добавление элемента в верхушку стека (заталкивание элемента)
	Удаление и восстановление элемента из верхушки стека (выталкивание элемента).

6. Ниже приведен код реализации стека в виде массива, но это влияет только на определение структуры и детали определения функций. Реализация не влияет на интерфейс, описанный прототипами функций.

```
/* stack.h -- интерфейс стека */
#include <stdbool.h>
/* ЗДЕСЬ ВСТАВЬТЕ ТИП ЭЛЕМЕНТА */
/* НАПРИМЕР, typedef int Item; */

#define MAXSTACK 100

typedef struct stack
{
   Item items[MAXSTACK]; /* содержит сведения о стеке          */
   int top;             /* индекс первой пустой ячейки        */
} Stack;

/* операция:         инициализация стека                       */
/* предусловия:      ps указывает на стек                      */
/* постусловия:      стек инициализирован пустым значением     */
void InitializeStack(Stack * ps);

/* операция:         проверяет, является ли стек полным        */
/* предусловия:      ps указывает на ранее инициализированный стек */
/* постусловия:      возвращает значение true, если стек полон, */
/*                   иначе возвращает значение false           */
bool FullStack(const Stack * ps);

/* операция:         проверяет, является ли стек пустым        */
/* предусловия:      ps указывает на ранее инициализированный стек */
/* постусловия:      возвращает значение true, если стек пуст, */
/*                   иначе возвращает значение false           */
bool EmptyStack(const Stack *ps);

/* операция:         заталкивает элемент в стек                */
/* предусловия:      ps указывает на ранее инициализированный стек */
/*                   элемент должен помещаться                 */
/*                   в верхушку стека                          */
/* постусловия:      если стек не полон, элемент помещается    */
/*                   в верхушку стека и функция возвращает     */
/*                   значение true; иначе стек остается        */
/*                   неизменным, а функция возвращает          */
/*                   значение false                            */
bool Push(Item item, Stack * ps);

/* операция:         удаляет элемент из верхушки стека         */
/* предусловия:      ps указывает на ранее инициализированный стек */
/* постусловия:      если стек не пуст, элемент в верхушке     */
/*                   стека копируется в *pitem и удаляется     */
/*                   из стека, а функция возвращает            */
/*                   значение true; если операция              */
/*                   опустошает стек, стек                     */
/*                   переустанавливается в пустое состояние.   */
/*                   Если стек пуст с самого начала, он        */
/*                   остается неизменным, а функция            */
/*                   возвращает значение false                 */
bool Pop(Item *pitem, Stack * ps);
```

6. Ниже приведено максимальное количество требуемых сравнений:

Количество элементов	Последовательный поиск	Двоичный поиск
3	3	2
1023	1023	10
65535	65535	16

7. Ответ показан на рис. А.1.

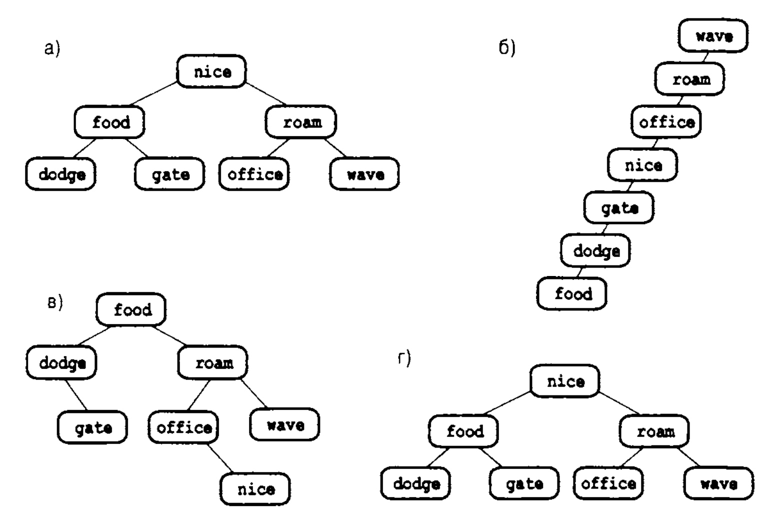


Рис. А.1. Двоичное дерево поиска слов

8. Ответ показан на рис. А.2.

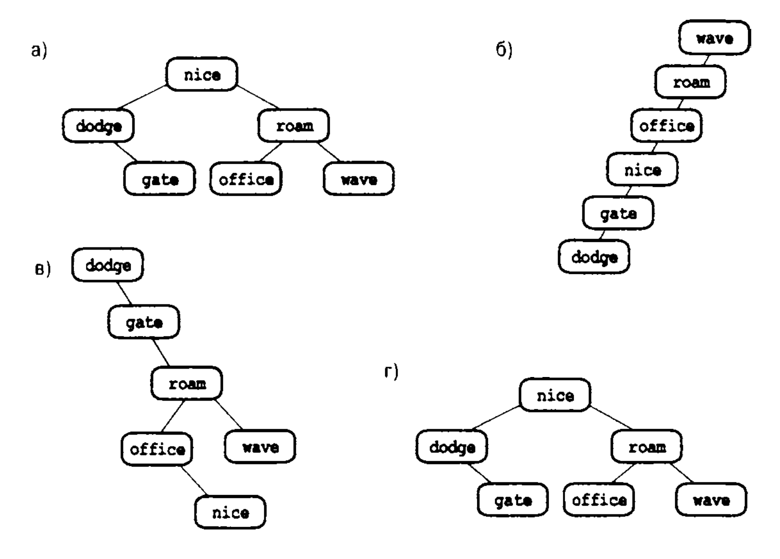


Рис. А.2. Двоичное дерево поиска слов после удаления

Б

Справочные материалы

В этой части книги предоставлен итоговый перечень базовых средств С наряду с более детализированным рассмотрением определенных тем. Ниже приведен список разделов данного приложения.

- Раздел I. Дополнительные источники информации
- Раздел II. Операции в языке С
- Раздел III. Базовые типы и классы хранения
- Раздел IV. Выражения, операторы и поток управления программы
- Раздел V. Стандартная библиотека ANSI С с дополнениями С99 и С11
- Раздел VI. Расширенные целочисленные типы
- Раздел VII. Расширенная поддержка символов
- Раздел VIII. Расширенные вычислительные средства С99/С11
- Раздел IX. Отличия между С и С++

Раздел I. Дополнительные источники информации

Если вы хотите узнать больше о языке C и программировании, вам будут полезны следующие ссылки.

Онлайновые ресурсы

Программисты на C помогали создавать Интернет, и теперь Интернет может помочь вам в изучении C. Интернет постоянно растет и изменяется; перечисленные ниже ресурсы – это пример того, что доступно на время написания книги. Разумеется, вы можете найти и другие онлайновые ресурсы.

В качестве возможного места старта, если у вас есть специфические вопросы о C, или же вы хотите расширить свои знания, обратитесь на сайт C FAQ (Frequently Asked Questions – часто задаваемые вопросы):

```
c-faq.com
```

Тем не менее, на указанном сайте раскрываются главным образом аспекты языка только до версии C89.

Если у вас есть вопросы по библиотеке C, то соответствующую информацию можно найти на следующем сайте:

```
www.acm.uiuc.edu/webmonkeys/book/c_guide/index.html
```

По приведенному ниже адресу предлагается всестороннее обсуждение указателей:

```
pweb.netcom.com/~tjensen/ptr/pointeБ.htm
```

Можете также использовать поисковые механизмы, чтобы найти статьи и сайты интересующей тематики:

```
www.google.com
search.yahoo.com
www.bing.com
```

С помощью расширенных средств поиска, предоставляемых перечисленными сайтами, можно более точно задать параметры поиска. Например, попробуйте поискать руководства по языку C.

Группы новостей позволяют задавать вопросы через Интернет. Обычно группы новостей доступны посредством программ чтения новостей, которые работают через учетную запись, предоставляемую поставщиком Интернет-услуг. К ним можно также получить доступ через веб-браузер, проследовав по адресу `http://groups.google.com`.

Вам должны сначала посвятить некоторое время чтению групп новостей, чтобы составить представление о том, какие темы они раскрывают. Например, если имеются вопросы о том, как сделать что-либо на C, поищите ответы в следующих группах новостей:

```
comp.lang.c
comp.lang.c.moderated
```

Здесь вы найдете людей, готовых и желающих помочь. Вопросы должны касаться стандартного языка C. Не спрашивайте о том, как организовать небуферизированный ввод в Unix – для этого предусмотрены специализированные группы новостей, посвященные специфичным для платформ вопросам. И никогда не спрашивайте, как вам справиться с домашними проблемами!

Если у вас возник вопрос об интерпретации стандарта C, попробуйте задать его в такой группе:

comp.std.c

Но не задавайте здесь вопросы о том, как объявлять указатель на трехмерный массив; такого рода вопрос больше подходит для группы comp.lang.c.

Наконец, если вы интересуетесь историей C, то Деннис Ритчи, создатель C, описал происхождение и разработку языка в статье по следующему адресу:

cm.bell-labs.com/cm/cs/who/dmr/chist.html

Книги по языку C

- Feuer, Alan R. *The C Puzzle Book, Revised Printing.* Upper Saddle River, NJ: Addison-Wesley Professional, 1998.

  Эта книга содержит множество программ, вывод из которых вы должны быть способны предсказать. Предсказание вывода дает хорошую возможность проверить и расширить свое понимание языка C. Эта книга также включает ответы и объяснения.

- Брайан У. Керниган, Деннис М. Ритчи. *Язык программирования C, 2-е издание.* ИД "Вильямс", 2014.

  Это второе издание первой книги о языке C. (Обратите внимание, что одним из авторов является Деннис Ритчи, создатель C.) В первом издании было представлено определение "K&R" C – неофициальный стандарт, существовавший на протяжении многих лет. Второе издание включает изменения ANSI, основанные на черновом варианте комитета ANSI, который был стандартом на время написания книги. Книга содержит множество интересных примеров. Однако в ней предполагается, что читатель знаком с системным программированием.

- Koenig, Andrew. *C Traps and Pitfalls. Reading*, MA: Addison-Wesley, 1989.

  Название ("Капканы и ловушки C") должно говорить само за себя.

- Summit, Steve. *C Programming FAQs. Reading*, MA: Addison-Wesley, 1995.

  Это расширенная версия часто задаваемых вопросов, доступных в Интернете.

Книги по программированию

- Kernighan, Brian W. and P.J. Plauger. *The Elements of Programming Style, Second Edition.* New York: McGraw-Hill, 1978.

  Эта книга содержит тонкие, ранее не издававшиеся классические эскизы примеров, собранные из других текстов для иллюстрации того, что нужно, и чего не нужно делать для получения ясных и эффективных программ.

- Кнут, Дональд Э. *Искусство программирования, том 1. Основные алгоритмы, 3-е издание.* ИД "Вильямс", 2000.

  В этом обширном классическом руководстве во всех подробностях рассматриваются представления данных и приводится анализ алгоритмов. По своей природе оно весьма глубокое и математическое. Том 2 (*Получисленные методы*; ИД "Вильямс", 2000 г.) включает расширенное обсуждение темы псевдослучайных чисел. Том 3 (*Сортировка и поиск*; ИД "Вильямс", 2000 г.), как следует из названия, посвящен вопросам сортировки и поиска. Примеры в книгах представлены с помощью псевдокода и на языке ассемблера.

- Sedgewick, Robert. Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching, Third Edition. Reading, MA: Addison-Wesley Professional, 1997.

  Как и можно было ожидать, книга посвящена структурам данных, сортировке и поиску.

Справочные руководства

- Harbison, Samuel P. and Steele, Guy L. *C: A Reference Manual, Fifth Edition.* Englewood Cliffs, NJ: Prentice Hall, 2002.

  В этом справочном руководстве представлены правила языка C и описана большая часть стандартных библиотечных функций. Оно включает обсуждение C99 и множество примеров.

- Plauger, P.J. *The Standard C Library.* Englewood Cliffs, NJ: Prentice Hall, 1992.

  В этом огромном справочном руководстве описаны стандартные библиотечные функции, но с более подробными объяснениями, чем можно найти в типовом руководстве по компилятору.

- *The International C Standard. ISO/IEC 9899:1999.*

  На момент написания книги этот стандарт доступен для загрузки за $285 из сайта `www.ansi.org` или за €238 из сайта Международной электротехнической комиссии. Не рассчитывайте изучить C по этому документу, поскольку он не задумывался как учебное пособие. Вот лишь одно довольно красноречивое утверждение из него: "Если в любом месте внутри единицы трансляции видимым является более одного объявления отдельного идентификатора, то синтаксический контекст устраняет неоднозначность случаев использования, ссылаясь на разные сущности".

Книги по C++

- Стивен Прата. *Язык программирования C++. Лекции и упражнения, 6-е издание.* ИД "Вильямс", 2014.

  Эта книга представляет собой введение в язык C++ и философию объектно-ориентированного программирования.

- Stroustrup, Bjarne. *The C++ Programming Language, Fourth Edition.* Reading, MA: Addison-Wesley, 2013.

  Книга, написанная создателем C++, представляет стандарт C++11.

- Бьярне Страуструп. *Программирование: принципы и практика использования C++, 2-е изд.*, ИД "Вильямс", 2015.

  Вводный курс программирования, написанный автором языка C++.

Раздел II. Операции в языке C

Язык C богат операциями. В табл. Б.II.1 перечислены операции C в порядке убывания приоритетов с указанием направления ассоциации. Все операции являются бинарными (с двумя операндами), если только не указано иначе. Обратите внимание, что некоторые бинарные и унарные операции, такие как * (умножение) и * (разыменование), обозначаются одним и тем же символом, но имеют разный приоритет. После таблицы приводятся краткие описания всех операций.

Таблица Б.II.1. Операции в С

Операции (в порядке снижения приоритетов)	Ассоциативность
++ (постфиксная форма) -- (постфиксная форма) () (вызов функции) [] {} (составной литерал) . ->	Слева направо
++ (префиксная форма) -- (префиксная форма) - + ~ ! * (разыменование) & (взятие адреса)	Слева направо
sizeof _Alignof (*тип*) (все унарные)	Справа налево
(*имя типа*)	Справа налево
* / %	Слева направо
+ - (обе бинарные)	Слева направо
<< >>	Слева направо
< > <= >=	Слева направо
== !=	Слева направо
&	Слева направо
^	Слева направо
|	Слева направо
&&	Слева направо
||	Слева направо
?: (условная операция)	Справа налево
= *= /= %= += -= <<= >>= &= |= ^=	Справа налево
, (операция запятой)	Слева направо

Арифметические операции

+ прибавляет значение справа к значению слева.

+ как унарная операция, дает значение, равное модулю операнда справа (с тем же знаком).

- вычитает значение справа из значения слева.

- как унарная операция, дает значение, равное модулю операнда справа (с противоположным знаком).

* умножает значение справа на значение слева.

/ делит значение слева на значение справа. Если оба операнда целочисленные, результат усекается до целого.

% дает остаток от целочисленного деления значения слева на значение справа (только для целых чисел).

++ добавляет 1 к значению переменной справа (в префиксном режиме), либо прибавляет 1 к значению переменной слева (в постфиксном режиме).

-- подобна ++, но вычитает 1.

Операции отношений

Каждая из следующих операций сравнивает значение слева от нее со значением справа:

<	меньше чем
<=	меньше или равно
==	равно
>=	больше или равно
>	больше чем
!=	не равно

Выражения отношений

Простейшее выражение отношения состоит из операции отношения с двумя операндами. Если сравнение истинно, выражение отношения имеет значение 1, а если ложно — то значение 0. Ниже показаны два примера:

5 > 2	истинно, имеет значение 1.
(2 + a) == a	ложно, имеет значение 0.

Операции присваивания

Язык С имеет одну базовую и несколько объединенных операций присваивания. Базовая форма записывается как одиночный знак равенства:

= присваивает значение справа l-значению слева.

Каждая из следующих операций присваивания обновляет l-значение, находящееся слева, значением, указанным справа, с использованием указанной операции (правый операнд обозначается как П-О , а левый — как Л-О):

+=	добавляет величину П-О к переменной Л-О и помещает результат в переменную Л-О.
-=	вычитает величину П-О из переменной Л-О и помещает результат в переменную Л-О.
*=	умножает переменную Л-О на величину П-О и помещает результат в переменную Л-О.
/=	делит переменную Л-О на величину П-О и помещает результат в переменную Л-О.
%=	получает остаток от деления величины Л-О на величину П-О и помещает результат в переменную Л-О.
&=	присваивает Л-О & П-О величине Л-О и помещает результат в переменную Л-О.
|=	присваивает Л-О | П-О величине Л-О и помещает результат в переменную Л-О.

`^=`	присваивает Л-О ^ П-О величине Л-О и помещает результат в переменную Л-О.
`>>=`	присваивает Л-О >> П-О величине Л-О и помещает результат в переменную Л-О.
`<<=`	присваивает Л-О << П-О величине Л-О и помещает результат в переменную Л-О.

Пример

`rabbits *= 1.6;` дает тот же результат, что и `rabbits = rabbits * 1.6;`

Логические операции

Логические операции обычно принимают в качестве операндов выражения отношений. Операция `!` принимает один операнд, остальные – два: левый и правый.

`&&`	И
`||`	ИЛИ
`!`	НЕ

Логические выражения

`выражение1 && выражение2`	истинно тогда и только тогда, когда оба выражения истинны.
`выражение1 || выражение2`	истинно, когда любое из выражений либо оба сразу истинны.
`!выражение`	истинно, когда выражение ложно, и наоборот.

Порядок оценки логических выражений

Логические выражения оцениваются слева направо. Оценка прекращается, как только становится ясно, что выражение ложно.

Примеры

`6 > 2 && 3 == 3`	истинно.
`! ( 6 > 2 && 3 == 3 )`	ложно.
`x != 0 && 20/x < 5`	второе выражение оценивается, только если x отлично от 0.

Условная операция

Операция `?:` принимает три операнда, каждый из которых является выражением. Они располагаются следующим образом:

```
выражение1 ? выражение2 : выражение3
```

Значение полного выражения равно значению *выражение2*, если *выражение1* истинно, и значению *выражение3* в противном случае.

Примеры

`( 5 > 3 ) ? 1 : 2`	имеет значение 1.

`( 3 > 5 ) ? 1 : 2` имеет значение 2.

`( a > b ) ? a : b` имеет значение, большее среди a и b.

Операции, связанные с указателями

& операция взятия адреса. Когда за ней следует имя переменной, & дает ее адрес.

* операция разыменования. Когда за ней следует указатель, * дает значение, сохраненное по указанному адресу.

Примеры

Здесь &nurce — это адрес переменной nurce:

```
nurse = 22;
ptr = &nurse;  /* указатель на nurse */
val = *ptr;
```

Общий результат заключается в присваивании переменной val значения 22.

Операции со знаком

- знак минуса меняет знак операнда на противоположный.

+ знак плюса оставляет знак операнда без изменений.

Операции структур и объединений

Структуры и объединения используют операции для идентификации их индивидуальных членов. Операция членства применяется со структурами и объединениями, а операция косвенного членства — с указателями на структуры и объединения.

Операция членства

Операция членства (.) используется с именем структуры или объединения для указания члена этой структуры или объединения. Если name — имя структуры или объединения, а member — член, определенный шаблоном структуры, то name.member идентифицирует этот член структуры. Типом name.member является тип, указанный для member. В аналогичной манере операция членства может применяться с объединениями.

Пример

```
struct {
   int code;
   float cost;
} item;
item.code = 1265;
```

Приведенный оператор присваивает значение элементу code структуры item.

Операция косвенного членства (или операция указателя на структуру)

Операция косвенного членства (->) используется с указателем на структуру или объединение для идентификации члена этой структуры или объединения. Предположим,

что ptrstr — указатель на структуру, а member — член, определенный шаблоном структуры. Тогда ptrstr->member идентифицирует член структуры, на которую ссылается указатель. В аналогичной манере операция косвенного членства может применяться с объединениями.

Пример

```
struct {
      int code;
      float cost;
} item, * ptrst;
ptrst = &item;
ptrst->code = 3451;
```

Этот фрагмент кода присваивает значение члену code структуры item. Следующие три выражения эквивалентны:

```
ptrst->code  item.code  (*ptrst).code
```

Побитовые операции

Все описанные ниже побитовые операции за исключением ~ являются бинарными.

~ унарная операция "НЕ", дающая в результате значение операнда, в котором каждый бит инвертирован.

& операция "И", дающая в результате значение, в котором каждый бит установлен в 1, если соответствующие биты в обоих операндах равны 1.

| операция "ИЛИ", дающая в результате значение, в котором каждый бит установлен в 1, когда любой из соответствующих битов операндов либо оба сразу равны 1.

^ операция исключающего "ИЛИ", дающая в результате значение, в котором каждый бит установлен в 1, когда любой из соответствующих битов операндов (но не оба сразу) равны 1.

<< операция сдвига влево, дающая значение, которое получено в результате сдвига битов левого операнда влево на количество позиций, указанное правым операндом. Освобождаемые места заполняются нулями.

>> операция сдвига вправо, дающая значение, которое получено в результате сдвига битов левого операнда вправо на количество позиций, указанное правым операндом. Для беззнаковых целых освобождаемые места заполняются нулями. Поведение для целых со знаком зависит от реализации.

Примеры

Предположим, что имеются следующие операторы:

```
int x = 2;
int y = 3;
```

Тогда x & y дает в результате значение 2, потому что только один бит "включен" как в x, так и в y. Кроме того, x << y дает в результате значение 12, поскольку это значение получается, когда битовый шаблон 3 сдвигается на 2 бита влево.

Прочие операции

Операция sizeof возвращает размер операнда, находящегося справа, который измерен в единицах, представляющих собой размер значения char. Обычно размер значения char составляет 1 байт. Операнд может быть спецификатором типа в круглых скобках, как в случае sizeof(float), или же именем определенной переменной, массива и тому подобного, как в случае sizeof foo. Типом выражения sizeof является size_t.

Операция _Alignof (C11) выдает требование к выравниванию для типа, указанного в операнде. В некоторых системах значения определенного типа должны сохраняться по адресам, кратным какой-то величине, такой как 4. Это целое число является требованием к выравниванию.

Операция (*тип*) — это операция приведения, которая преобразует следующее за ней значение в тип, указанный с помощью ключевого слова в круглых скобках. Например, (float) 9 преобразует целое число 9 в число с плавающей запятой 9.0.

Операция , — это операция запятой, которая связывает два выражения в одно и гарантирует, что левое выражение будет оценено первым. Значением всего результирующего выражения является значение правого выражения. Операция запятой обычно используется для включения большего количества информации в управляющее выражение цикла for.

Пример

```
for (step = 2, fargo = 0; fargo < 1000; step *= 2)
    fargo += step;
```

Раздел III. Базовые типы и классы хранения

Сводка: базовые типы данных

Базовые типы данных C подразделяются на две категории: целые числа и числа с плавающей запятой. Различные вариации характеризуются разными диапазонами значений и точностью.

Ключевые слова

Базовые типы данных устанавливаются с применением следующих восьми ключевых слов: int, long, short, unsigned, char, float, double и signed (ANSI C).

Целые числа со знаком

Целые числа со знаком могут иметь положительные и отрицательные значения.

- int — базовый целочисленный тип для данной системы.
- long или long int — может содержать целое число, как минимум, равное самому большому значению int, и возможно больше; long занимает не менее 32 битов.
- Самое большое значение short или short int не больше самого большого int, но может быть меньше. short занимает минимум 16 битов. Обычно long больше, чем short, а int — такой же, как один из них. Например, компиляторы C для DOS на IBM PC поддерживают 16-битовые int и short и 32-битовый long.
- Тип long long, предусмотренный стандартом C99, имеет размер не меньше, чем у long, и занимает минимум 64 бита.

Целые числа без знака

Целые числа без знака могут иметь только нулевое или положительные значения, что расширяет диапазон допустимых положительных чисел. Используйте ключевое слово unsigned перед именем желаемого типа: unsigned int, unsigned long, unsigned short или unsigned long long. Указание только одного ключевого слова unsigned означает то же самое, что и unsigned int.

Символы

Символы — это типографские знаки, такие как A, & и +. По определению для переменной типа char используется один байт памяти. В прошлом наиболее типичным был размер char, равный 8 битам. Однако возможность языка С по обработке расширенных наборов символов может привести к применению 16-битовых и даже 32-битовых символов.

Ключевым словом для обозначения этого типа является char. Некоторые реализации используют char со знаком, другие — char без знака. ANSI С позволяет применять ключевые слова signed и unsigned для указания требуемой формы char. Формально char, unsigned char и signed char — три разных типа, причем тип char имеет такое же представление, как один из двух других.

Булевский тип (C99)

Булевским типом в C99 является _Bool. Это целочисленный тип без знака, который может принимать одно из двух значений: 0 для обозначения лжи и 1 — для истины. Включение заголовочного файла stdbool.h позволяет использовать bool вместо _Bool, true — вместо 1 и false — вместо 0, что обеспечивает совместимость кода с С++.

Вещественные и комплексные типы с плавающей запятой

Стандарт C99 выделяет две области типов с плавающей запятой: вещественные и комплексные. Вместе они образуют типы с плавающей запятой.

Вещественные числа с плавающей запятой могут иметь положительные и отрицательные значения. В языке С распознаются три вещественных типа с плавающей запятой.

- float — базовый тип с плавающей запятой. Может представлять как минимум шесть значащих цифр. Обычно занимает 32 бита.
- double — (возможно) более крупная единица хранения чисел с плавающей запятой. Может допускать больше значащих цифр и, возможно, бóльшие экспоненты, чем тип float. Может представлять, по меньшей мере, 10 значащих цифр. Обычно занимает 64 бита.
- long double — (возможно) еще более крупная единица хранения чисел с плавающей запятой. Может допускать больше значащих цифр и, возможно, бóльшие экспоненты, чем тип double.

Комплексные числа имеют два компонента: действительную часть и мнимую часть. Стандарт C99 внутренне представляет комплексное число в виде двухэлементного массива, в котором первый элемент является действительной частью, а второй — мнимой частью. Существуют три типа комплексных чисел.

- float _Complex — представляет действительную и мнимую части с помощью значений типа float.

- double _Complex — представляет действительную и мнимую части с помощью значений типа double.
- long double _Complex — представляет действительную и мнимую части с помощью значений типа long double.

В каждом случае тип в префиксе называется *соответствующим вещественным типом.* Например, соответствующим вещественным типом в double_Complex является double.

В стандарте C99 комплексные типы были необязательными для автономных сред, в которых программы С могли выполняться без операционной системы. В стандарте C11 комплексные типы необязательны и для автономных, и для размещаемых сред.

Существуют также три мнимых типа; они необязательны как для автономных сред, так и для размещаемых сред (сред, в которых программы С выполняются под управлением операционной системы). Мнимое число имеет только мнимую часть.

Ниже приведен список мнимых типов.

- float _Imaginary — представляет мнимую часть с помощью значений типа float.
- double _Imaginary — представляет мнимую часть с помощью значений типа double.
- long double _Imaginary — представляет мнимую часть с помощью значений типа long double.

Комплексные числа могут быть инициализированы с применением вещественных чисел и значения I, которое определено в complex.h и представляет i, т.е. квадратный корень из -1:

```
#include <complex.h>  // для I
double _Complex z = 3.0;       // действительная часть = 3.0, мнимая часть = 0
double _Complex w = 4.0 * I;  // действительная часть = 0.0, мнимая часть = 4.0
double Complex u = 6.0 - 8.0 * I; //действительная часть = 6.0, мнимая часть = -8.0
```

Библиотека complex.h, обсуждаемая позже в этом приложении, включает функции, которые возвращают действительный и мнимый компоненты комплексного числа.

Сводка: объявление простой переменной

1. Выберите необходимый тип.
2. Выберите имя для переменной.
3. Используйте следующий формат оператора объявления:

   спецификатор-типа имя-переменной;

   спецификатор-типа формируется из одного или более ключевых слов. Вот некоторые примеры:

   ```
   int erest;
   unsigned short cash;
   ```

4. Чтобы объявить несколько переменных одного и того же типа, отделяйте их имена друг от друга запятыми:

   ```
   char ch, init, ans;
   ```

5. В операторе объявления переменную можно инициализировать:

   ```
   float mass = 6.0E24;4
   ```

Сводка: классы хранения

Ключевые слова

auto, extern, static, register, _Thread_local (C11)

Общий комментарий

Класс хранения переменной определяет ее область видимости, связывание и продолжительность хранения. Класс хранения задается как местом ее определения, так и указанными в определении ключевыми словами. Переменные, определенные вне всех функций, являются внешними, имеют область видимости в пределах файла, внешнее связывание и статическую продолжительность хранения. Переменные, определенные внутри какой-то функции, являются автоматическими, если только не использовано одно из других ключевых слов. Они характеризуются областью видимости в пределах блока, отсутствием связывания и автоматической продолжительностью хранения. Переменным, определенным с ключевым словом static внутри функции, присуща область видимости в пределах блока, отсутствие связывания и статическая продолжительность хранения. Переменные, определенные с ключевым словом static вне функции, имеют область видимости в пределах файла, внутреннее связывание и статическую продолжительность хранения. В стандарте C11 появился новый квалификатор класса хранения: _Thread_local. Объявленный с этим квалификатором объект имеет потоковую продолжительность хранения, т.е. он существует в течение времени жизни потока, в котором объявлен, и инициализируется, когда поток начинается. Таким образом, объект подобного рода является локальным по отношению к потоку.

Свойства

Ниже представлена сводка по свойствам классов хранения.

Класс хранения	Продолжительность хранения	Область видимости	Связывание	Как объявляется
Автоматический	Автоматическая	Блок	Нет	В блоке
Регистровый	Автоматическая	Блок	Нет	В блоке с ключевым словом register
Статический с внешним связыванием	Статическая	Файл	Внешнее	Вне всех функций
Статический с внутренним связыванием	Статическая	Файл	Внутреннее	Вне всех функций с ключевым словом static
Статический без связывания	Статическая	Блок	Нет	В блоке ключевым словом static
Потоковый с внешним связыванием	Потоковая	Файл	Внешнее	Вне всех блоков с ключевым словом _Thread_local
Потоковый с внутренним связыванием	Потоковая	Файл	Внутреннее	Вне всех блоков с ключевыми словами static и _Thread_local
Потоковый без связывания	Потоковая	Блок	Нет	Внутри блока с ключевыми словами static и _Thread_local

Обратите внимание, что ключевое слово extern применяется только для повторного объявления переменной, которая была определена внешне где-то в другом месте. Объявление переменной за пределами функции делает ее внешней.

В дополнение к этим классам хранения язык C предоставляет выделенную память. Такая память выделяется вызовом одной из функций семейства malloc(), возвращающей указатель, который может быть использован для доступа к памяти. Память остается выделенной до тех пор, пока не будет вызвана функция free() либо не завершится работа программы. Доступ к выделенной подобным образом памяти возможен из любой функции, которая располагает соответствующим указателем. Например, функция может передать значение указателя другой функции, давая ей возможность доступа к памяти.

Сводка: квалификаторы

Ключевые слова

Для квалификации переменных применяются следующие ключевые слова:

```
const, volatile, restrict
```

Общий комментарий

Квалификатор определенным образом ограничивает использование переменной. Переменная const после инициализации не может быть изменена. Компилятор не может предполагать, что переменная volatile не изменяется каким-то внешним действием, таким как аппаратное обновление. Указатель, квалифицированный с помощью restrict, понимается как обеспечивающий единственный доступ (в определенной области видимости) к блоку памяти.

Примеры

Объявление

```
const int joy = 101;
```

устанавливает, что значение joy зафиксировано как 101.

Объявление

```
volatile unsigned int incoming;
```

устанавливает, что значение переменной incoming может измениться между несколькими ее упоминаниями в программе.

Объявление

```
const int * ptr = &joy;
```

устанавливает, что указатель ptr не может быть использован для изменения переменной joy. Однако указатель может быть переустановлен, чтобы ссылаться на другую ячейку памяти.

Объявление

```
int * const ptr = &joy;
```

устанавливает, что указатель ptr не может изменяться, т.е. он может указывать только на joy. Однако он может применяться для изменения значения joy.

Прототип

```
void simple(const char * s);
```

устанавливает, что после инициализации формального аргумента s любым переданным функции simple() значением эта функция не может изменять значение, на которое указывает s.

Прототип

```
void supple(int * const pi);
```

и эквивалентный ему прототип

```
void supple(int pi[const]);
```

устанавливают, что функция `supple()` не может изменять значение параметра `pi`.

Прототип

```
void interleave(int * restrict p1, int * restrict p2, int n);
```

устанавливает, что `p1` и `p2` являются единственным первичным средством доступа к блокам памяти, на которые они указывают; это подразумевает, что данные два блока не перекрываются.

Раздел IV. Выражения, операторы и поток управления программы

Сводка: выражения и операторы

В языке С выражения представляют значения, а операторы – инструкции, исполняемые компьютером.

Выражения

Выражение – это комбинация операций и операндов. Простейшее выражение – это всего лишь константа или переменная без каких-либо операций, такая как `22` или `beebop`. Более сложные примеры могут выглядеть так: `55+22` и `vap=2*(vip+(vup=4))`.

Операторы

Оператор – это команда компьютеру. Любое выражение, за которым следует точка с запятой, формирует оператор, хотя и не обязательно осмысленный. Операторы могут быть простыми или составными. *Простые операторы* завершаются точкой с запятой, как показано в следующих примерах:

Оператор объявления	`int toes;`
Оператор присваивания	`toes = 12;`
Оператор вызова функции	`printf ("%d\n", toes);`
Управляющий оператор	`while (toes < 20) toes = toes + 2;`
Пустой оператор	`; /* ничего не делает */`

(Формально стандарт относит объявления к отдельной категории, а не объединяет их с операторами.)

Составные операторы, или *блоки*, состоят из одного или более операторов (каждый из которых сам может быть составным), заключенных в фигурные скобки. Примером может служить следующий оператор `while`:

```
while (years < 100)
{
   wisdom = wisdom + 1;
   printf("%d %d\n", years, wisdom);
   years = years + 1;
}
```

Сводка: оператор while

Ключевое слово

Ключевым словом оператора while является while.

Общий комментарий

Оператор while создает цикл, который повторяется до тех пор, пока проверяемое *выражение* не станет равно false или нулю. Оператор while представляет собой цикл с *проверкой условия на входе*; решение о выполнении очередной итерации принимается *перед* выполнением тела цикла. Таким образом, существует возможность, что тело цикла не будет выполнено ни разу. Часть *оператор* этой формы может быть как простым оператором, так и составным.

Форма

```
while(выражение)
   оператор
```

Часть *оператор* повторяется до тех пор, пока *выражение* не станет ложным или равным нулю.

Примеры

```
while (n++ < 100)
   printf(" %d %d\n",n, 2*n+1);

while (fargo < 1000)
{
   fargo = fargo + step;
   step = 2 * step;
}
```

Сводка: оператор for

Ключевое слово

Ключевым словом оператора for является for.

Общий комментарий

В операторе for для управления циклическим процессом используются три управляющих выражения, разделенные точками с запятой. Выражение *инициализация* выполняется один раз перед любыми другими операторами цикла. Если выражение *проверка* является истинным (или ненулевым), выполняется одна итерация цикла. Затем оценивается выражение *обновление*, после чего вновь оценивается выражение *проверка*. Оператор while представляет собой цикл с *проверкой условия на входе*; решение о выполнении очередной итерации принимается *перед* выполнением тела цикла. Таким образом, вполне возможно, что цикл не выполнится ни разу. Часть *оператор* может быть как простым оператором, так и составным.

Форма

```
for (инициализация ; проверка ; обновление)
   оператор
```

Цикл повторяется до тех пор, пока выражение *проверка* не окажется ложным или равным нулю.

Стандарт C99 позволяет включать объявление в часть *инициализация*. Область видимости и продолжительность хранения переменной ограничены циклом `for`.

Примеры

```
for (n = 0;  n < 10 ; ++n)
   printf("%d %d\n", n, 2 * n+1);

for (int k = 0;  k < 10 ; ++k)        // C99
   printf("%d %d\n", k, 2 * k+1);
```

Сводка: оператор do while

Ключевые слова

Ключевыми словами оператора `do while` являются `do` и `while`.

Общий комментарий

Оператор `do while` создает цикл, повторяющийся до тех пор, пока проверочное *выражение* не станет ложным или равным нулю. Оператор `do while` является циклом с *проверкой условия на выходе*, решение о выполнении очередной итерации принимается *после* выполнения тела цикла. Таким образом, цикл должен выполниться как минимум один раз. Часть *оператор* этой формы цикла может быть как одиночным оператором, так и составным.

Форма

```
do
   оператор
while (выражение);
```

Часть *оператор* повторяется до тех пор, пока *выражение* не станет ложным или равным нулю.

Пример

```
do
   scanf("%d", &number)
while(number != 20);
```

Сводка: использование операторов if для реализации выбора

Ключевые слова

Ключевыми словами оператора `if` являются `if` и `else`.

Общий комментарий

В каждой из показанных ниже форм *оператор* может быть как одиночным оператором, так и составным. "Истинное" выражение в общем случае означает такое, которое дает в результате ненулевое значение.

Форма 1

```
if(выражение)
   оператор
```

Если *выражение* истинно, то выполняется *оператор*.

Форма 2

```
if(выражение)
   оператор1
else
   оператор2
```

Если *выражение* истинно, то выполняется *оператор1*. В противном случае выполняется *оператор2*.

Форма 3

```
if(выражение1)
   оператор1
else if(выражение2)
   оператор2
else
   оператор3
```

Если *выражение1* истинно, то выполняется *оператор1*. Если же *выражение1* ложно, но *выражение2* истинно, то выполняется *оператор2*. Иначе, если оба выражения ложны, выполняется *оператор3*.

Пример

```
if (legs == 4)
   printf("Это может быть лошадь.\n");
else if (legs > 4)
    printf("Это не лошадь.\n");
else   /* случай, когда legs < 4 */
{
   legs++;
   printf("Теперь на одну ногу стало больше.\n");
}
```

Сводка: множественный выбор с помощью switch

Ключевые слова

Ключевым словом оператора `switch` является `switch`.

Общий комментарий

Управление передается оператору, снабженному меткой *выражение*. Поток управления затем проходит остальные операторы внутри блока `switch`, если только снова не будет перенаправлен. Как *выражение*, так и метки `case` должны иметь целочисленные значения (включая тип `char`), а метки должны быть константами или выражениями, состоящими исключительно из констант. Если ни одна метка не соответствует значению выражения, управление переходит к оператору, помеченному меткой `default`, если это предусмотрено. Иначе управление переходит к оператору, следующему за оператором `switch`. После того, как управление передается по определенной метке, выполняются все последующие операторы внутри `switch`, до конца `switch` или до оператора `break` в зависимости от того, что встретится раньше.

Форма

```
switch (выражение)
{
   case метка1 : оператор1
   case метка2 : оператор2
   default     : оператор3
}
```

Операторов, снабженных метками, может быть больше двух, а конструкция default является необязательной.

Примеры

```
switch (value)
   case 1  : find_sum(ar, n);
             break;
   case 2  : show_array(ar, n);
             break;
   case 3  : puts("Всего хорошего!");
             break;
   default : puts("Неправильный выбор, попробуйте еще раз.");
             break;
}
switch (letter)
{
   case 'a' :
   case 'e' : printf("%d является гласной буквой\n", letter);
   case 'c' :
   case 'n' : printf("%d находится в слове \"cane\"\n", letter);
   default  : printf("Всего хорошего.\n");
}
```

Если letter имеет значение 'a' или 'e', то выводятся все три сообщения, а если 'c' или 'n' — то два последних сообщения. Все прочие значения приводят к выводу только последнего сообщения.

Сводка: переходы в программе

Ключевые слова

Ключевыми словами для переходов в программе являются break, continue и goto.

Общий комментарий

Эти три инструкции — break, continue и goto — заставляют поток управления программы переходить из одного места кода в другое.

Команда break

Команда break может использоваться с любой из трех форм циклов и с оператором switch. Она вынуждает поток управления программы пропустить остаток цикла или оператора switch, который ее содержит, и продолжить выполнение со следующей инструкции после цикла или switch.

Пример

```
while ((ch = getchar()) != EOF)
{
   putchar(ch);
   if (ch == ' ')
        break;      // прекратить выполнение цикла
   chcount++;
}
```

Команда continue

Команда continue может использоваться с любой из трех форм циклов, но не с оператором switch. Она заставляет поток управления программы пропустить оставшиеся операторы в цикле. В случае циклов for и while запускается следующая итерация. В случае цикла do while проверяется условие выхода, а затем, если необходимо, запускается новая итерация.

Пример

```
while ((ch = getchar()) != EOF)
{
    if (ch == ' ')
        continue;    // перейти к проверочному условию
    putchar(ch);
    chcount++;
}
```

Этот фрагмент кода отображает и подсчитывает непробельные символы.

Команда goto

Оператор goto передает управление оператору, снабженному указанной меткой. Метка отделяется от оператора двоеточием. Имена меток подчиняются правилам, которые регламентируют именование переменных. Помеченный оператор может располагаться как до, так и после goto.

Форма

```
goto метка;
метка : оператор
```

Пример

```
top : ch = getchar();
    if (ch != 'y')
goto top;
```

Раздел V. Стандартная библиотека ANSI C с дополнениями C99 и C11

Библиотека ANSI C классифицирует функции по нескольким группам, с каждой из которых ассоциирован свой заголовочный файл. В этом разделе представлен обзор библиотеки, список заголовочных файлов и краткое описание связанных с ними функций. Некоторые из этих функций (например, часть функций ввода-вывода) обсуждаются более подробно. За полным описанием обращайтесь к документации, сопровождающей вашу реализацию, к справочному руководству или же к онлайновому руководству наподобие http://www.acm.uiuc.edu/webmonkeys/book/c_guide/.

Диагностика: assert.h

В заголовочном файле assert.h определен макрос assert(). Определение идентификатора NDEBUG перед включением заголовочного файла assert.h делает макрос assert() неактивным. Выражение, используемое в качестве аргумента, обычно является выражением отношения или логическим, которое должно быть истинным в этой точке программы, если программа функционирует корректно. Макрос assert() описан в табл. Б.V.1.

Таблица Б.V.1. Макрос для диагностики

Прототип	Описание
`void assert(int exprs);`	Если выражение `exprs` оценивается как ненулевое (истинное), макрос ничего не делает. Если же `exprs` равно 0 (ложное), макрос `assert()` отображает выражение, номер строки, в которой находится оператор `assert()`, а также имя файла, содержащего этот оператор. Затем он вызывает функцию `abort()`

В стандарте С11 в библиотеку добавлен макрос `static_assert`, который расширяется до `_Static_assert`. В свою очередь, `_Static_assert` представляет собой ключевое слово, которое считается формой объявления. Оно обеспечивает проверку на этапе компиляции, которая применяется следующим образом:

```
_Static_assert (константное-выражение, строковый-литерал);
```

Если *константное-выражение* оценивается как имеющее значение 0, то компилятор выдает сообщение об ошибке, которое включает *строковый-литерал*. В противном случае никаких действий не предпринимается.

Комплексные числа: complex.h (C99)

В стандарте C99 была добавлена интенсивная поддержка вычислений с комплексными числами, а в стандарте С11 она дополнительно расширена. В реализациях допускается принимать решение о предоставлении типа `_Imaginary` вдобавок к типу `_Complex`. В стандарте С11 оба эти типа являются необязательными. В стандарте C99 тип `_Complex` был обязательным, а тип `_Imaginary` — необязательным. В разделе VIII этого приложения приводится дополнительное обсуждение поддержки комплексных типов. В заголовочном файле `complex.h` определены макросы, перечисленные в табл. Б.V.2.

Таблица Б.V.2. Макросы в complex.h

Прототип	Описание
`complex`	Расширяется до ключевого слова типа `_Complex`
`_Complex_I`	Расширяется до выражения типа `const float _Complex`, значение которого, возведенное в квадрат, равно -1
`imaginary`	Если поддерживаются мнимые типы, расширяется до ключевого слова `_Imaginary_I`
`_Imaginary_I`	Если поддерживаются мнимые типы, расширяется до выражения типа `const float _Imaginary_I`, значение которого, возведенное в квадрат, равно -1
`I`	Расширяется либо до `_Complex_I`, либо до `_Imaginary_I`

Реализация комплексных чисел в С, поддерживаемая заголовочным файлом `complex.h`, существенно отличается от их реализации в С++, поддерживаемой заголовочным файлом `complex`. Для определения типов комплексных чисел в языке С++ используются классы.

С помощью прагмы `STDC CX_LIMITED_RANGE` можно указать, разрешено ли применять обычные математические формулы (установка `on`), или же особое внимание должно уделяться предельным значениям (установка `off`):

```
#include <complex.h>
#pragma STDC CX_LIMITED_RANGE on
```

Библиотечные функции поставляются в трех разновидностях: double, float и long double. В табл. Б.V.3 перечислены функции для версии double. В версиях float и long double к именам функций добавляются, соответственно, f и l. То есть csinf() — это версия float функции csin(), a csinl() — версия long double той же функции. Углы измеряются в радианах.

Таблица Б.V.3. Функции для работы с комплексными числами

Прототип	Описание
double complex cacos(double complex z)	Возвращает комплексный арккосинус z
double complex casin(double complex z);	Возвращает комплексный арксинус z
double complex catan(double complex z);	Возвращает комплексный арктангенс z
double complex ccos(double complex z);	Возвращает комплексный косинус z
double complex csin(double complex z);	Возвращает комплексный синус z
double complex ctan(double complex z);	Возвращает комплексный тангенс z
double complex cacosh(double complex z);	Возвращает комплексный гиперболический арккосинус z
double complex casinh(double complex z);	Возвращает комплексный гиперболический арксинус z
double complex catanh(double complex z);	Возвращает комплексный гиперболический арктангенс z
double complex ccosh(double complex z);	Возвращает комплексный гиперболический косинус z
double complex csinh(double complex z);	Возвращает комплексный гиперболический синус z
double complex ctanh(double complex z);	Возвращает комплексный гиперболический тангенс z
double complex cexp(double complex z);	Возвращает комплексное значение *e* в степени z
double complex clog(double complex z);	Возвращает комплексный натуральный (по основанию *e*) логарифм z
double complex cabs(double complex z);	Возвращает абсолютное значение z
double complex cpows(double complex z, double complex y);	Возвращает значение z в степени y
double complex csqrt(double complex z);	Возвращает комплексный квадратный корень из z
double complex carg(double complex z);	Возвращает фазовый угол (или аргумент) z в радианах
double complex cimag(double complex z);	Возвращает мнимую часть z как вещественное число
double complex conj(double complex z);	Возвращает комплексное сопряжение z
double complex cproj(double complex z);	Возвращает проекцию z на сферу Римана
double complex CMPLX(double x, double y);	Возвращает комплексное число, действительным компонентом которого является x, а мнимым компонентом — y (C11)
double complex creal(double complex z);	Возвращает действительную часть z как вещественное число

Обработка символов: ctype.h

Эти функции принимают аргументы int, которые должны иметь возможность быть представленными либо как unsigned char, либо как EOF; в случае передачи других значений поведение не определено. В табл. Б.V.4 значение true используется в качестве синонима для ненулевого значения. Интерпретация некоторых определений зависит от текущих локальных установок, которые управляются функциями из заголовочного файла locale.h; таблица демонстрирует интерпретацию для локальной установки "C".

Таблица Б.V.4. Функции обработки символов

Прототип	Описание
int isalnum(int c);	Возвращает true, если c — алфавитно-цифровой символ (буква или цифра)
int isalpha(int c);	Возвращает true, если c — буква
int isblank(int c);	Возвращает true, если c — пробел или горизонтальная табуляция (C99)
int iscntrl(int c);	Возвращает true, если c — управляющий символ, такой как <Ctrl+B>
int isdigit(int c);	Возвращает true, если c — десятичная цифра
int isgraph(int c);	Возвращает true, если c — печатаемый символ, отличный от пробела
int islower(int c);	Возвращает true, если c — символ нижнего регистра
int isprint(int c);	Возвращает true, если c — печатаемый символ
int ispunct(int c);	Возвращает true, если c — знак препинания (любой печатаемый символ, отличный от пробела или алфавитно-цифрового символа)
int isspace(int c);	Возвращает true, если c — пробельный символ: пробел, новая строка, перевод страницы, возврат каретки, вертикальная табуляция, горизонтальная табуляция или, возможно, другой символ, определяемый реализацией
int isupper(int c);	Возвращает true, если c — символ верхнего регистра
int isxdigit(int c);	Возвращает true, если c — шестнадцатеричная цифра
int tolower(int c);	Если аргумент — символ верхнего регистра, возвращает его версию в нижнем регистре; иначе просто возвращает исходный аргумент
int toupper(int c);	Если аргумент — символ нижнего регистра, возвращает его версию в верхнем регистре; иначе просто возвращает исходный аргумент

Сообщение об ошибках: errno.h

Заголовочный файл errno.h поддерживает старый механизм сообщения об ошибках. Этот механизм предоставляет ячейку во внешней статической памяти, которая доступна через идентификатор (или, возможно, макрос) ERRNO. Некоторые библиотечные функции помещают в эту ячейку значение, чтобы сообщить об ошибке. Программа, включающая заголовочный файл errno.h, может затем проверить значение ERRNO, чтобы выяснить, возникла ли конкретная ошибка. Механизм, использующий ERRNO, считается устаревшим, и математические функции больше не обязаны устанавливать значения ERRNO. В стандарте предусмотрены три значения в виде макросов, которые представляют определенные ошибки, но конкретные реализации могут предоставлять их больше. В табл. Б.V.5 перечислены стандартные макросы.

Таблица Б.V.5. Макросы в errno.h

Прототип	Описание
EDOM	Ошибка предметной области в вызове функции (аргумент вышел за допустимые пределы)
ERANGE	Ошибка диапазона возвращаемого значения функции (возвращаемое значение вышло за пределы допустимого диапазона)
EILSEQ	Ошибка трансляции широких символов

Среда плавающей запятой: fenv.h (C99)

Стандарт C99 предоставляет доступ и возможность управления средой плавающей запятой через заголовочный файл fenv.h.

Среда плавающей запятой состоит из набора флагов состояния и режимов управления. Исключительные ситуации, возникающие во время вычислений с плавающей запятой, такие как деление на ноль, могут "генерировать исключение". Это означает, что событие устанавливает один из флагов среды плавающей запятой. Значение режима управления может управлять, например, направлением округления. В заголовочном файле fenv.h определен набор макросов, представляющих несколько исключений и режимов управления, а также прототипы функций, которые взаимодействуют со средой. Заголовок также предоставляет прагму для включения или отключения доступа к среде плавающей запятой.

Директива

```
#pragma STDC FENV_ACCESS on
```

включает доступ к этой среде, а директива

```
#pragma STDC FENV_ACCESS off
```

отключает его. Если прагма является внешней, она должна находиться перед любым внешним объявлением или же в начале составного блока. Она остается в силе до тех пор, пока не будет переключена другим экземпляром прагмы, либо до достижения конца файла (внешняя директива) или конца составного оператора (блочная директива).

В заголовочном файле fenv.h определены два типа, показанные в табл. Б.V.6.

Таблица Б.V.6. Типы в fenv.h

Тип	Описание
fenv_t	Вся среда плавающей запятой
fexcept_t	Коллекция флагов состояния среды плавающей запятой

В заголовочном файле fenv.h также определены макросы, представляющие несколько возможных исключений плавающей запятой и управляющих состояний. Реализации могут определять дополнительные макросы, назначая им имена, которые начинаются с FE_ и состоят из заглавных букв. В табл. Б.V.7 приведены стандартные макросы исключений.

Таблица Б.V.7. Макросы в fenv.h

Макрос	Описание
FE_DIVBYZERO	Исключение деления на ноль
FE_INEXACT	Исключение неточного значения
FE_INVALID	Исключение некорректного значения
FE_OVERFLOW	Исключение переполнения
FE_UNDERFLOW	Исключение потери значимости
FE_ALL_EXCEPT	Объединение с помощью операции побитового "ИЛИ" всех исключений плавающей запятой, поддерживаемых реализацией
FE_DOWNWARD	Округление в меньшую сторону
FE_TONEAREST	Округление до ближайшего значения
FE_TOWARDZERO	Округление в сторону нуля
FE_UPWARD	Округление в большую сторону
FE_DFL_ENV	Представляет стандартную среду и имеет тип const fenv_t *

В табл. Б.V.8 показаны прототипы стандартных функций из заголовочного файла fenv.h. Обратите внимание, что очень часто значения аргументов и возвращаемые значения соответствуют макросам из табл. Б.V.7. Например, FE_UPWARD является подходящим аргументом для fesetround().

Таблица Б.V.8. Прототипы в fenv.h

Прототип	Описание
void feclearexcept(int excepts);	Очищает исключения, представленные с помощью excepts
void fegetexceptflag (fexcept_t *flagp, int excepts);	Сохраняет состояния флагов среды плавающей запятой, заданные посредством excepts, в объекте, на который указывает flagp
void feraiseexcept(int excepts);	Генерирует исключения, указанные с помощью excepts
void fesetexceptflag (const fexcept_t *flagp, int excepts);	Устанавливает флаги состояния среды плавающей запятой, указанные в excepts, в значения, представленные flagp; параметр flagp должен быть установлен предыдущим вызовом fegetexceptflag()
int fetestexcept(int excepts);	В excepts указаны флаги состояния, которые следует опросить; функция возвращает объединенные побитовым "ИЛИ" значения заданных флагов состояния
int fegetround(void);	Возвращает текущее направление округления
int fesetround(int round);	Устанавливает направление округления в round; возвращает 0 только в случае успешного выполнения
void fegetenv(fenv_t *envp);	Сохраняет текущую среду в области памяти, на которую указывает envp
int feholdexcept(fenv_t *envp);	Сохраняет текущую среду плавающей запятой в области памяти, указанной envp, очищает флаги состояния, а затем, если возможно, устанавливает безостановочный режим, при котором выполнение продолжается, невзирая на исключения; возвращает 0 только в случае успешного выполнения

Окончание табл. Б.V.8

Прототип	Описание
`void fesetenv(const fenv_t *envp);`	Устанавливает среду плавающей запятой, представленную `envp`; параметр `envp` должен указывать на объект данных, установленный предыдущим вызовом `fegetenv()` или `feholdexcept()` либо макросом среды плавающей запятой
`void feupdateenv (const fenv_t *envp);`	Функция сохраняет текущие сгенерированные исключения плавающей запятой в автоматическом хранилище, устанавливает среду плавающей запятой, представленную объектом, на который указывает `envp`, и затем генерирует сохраненные исключения плавающей запятой; параметр `envp` должен указывать на объект данных, установленный предыдущим вызовом `fegetenv()` или `feholdexcept()` либо макросом среды плавающей запятой

Характеристики среды плавающей запятой: `float.h`

В заголовочном файле `float.h` определено несколько макросов, представляющих разнообразные пределы и параметры. Эти макросы перечислены в табл. Б.V.9; добавления, появившиеся в C11, выделены курсивом. Многие макросы имеют отношение к следующей модели представления с плавающей запятой:

$$x = sb^e \sum_{k=1}^{p} f_k b^k$$

Если самая первая цифра f_1 является ненулевой (и x отлично от нуля), то такое число называется *нормализованным числом с плавающей запятой*. Соответствующие объяснения приведены в разделе VIII этого приложения, в том числе и для ряда показанных макросов.

Таблица Б.V.9. Макросы в `fenv.h`

Макрос	Описание
`FLT_ROUNDS`	Стандартная схема округления
`FLT_EVAL_METHOD`	Стандартная схема для оценки выражений с плавающей запятой
`FLT_HAS_SUBNORM`	Наличие или отсутствие субнормальных чисел типа `float`
`DBL_HAS_SUBNORM`	Наличие или отсутствие субнормальных чисел типа `double`
`LDBL_HAS_SUBNORM`	Наличие или отсутствие субнормальных чисел типа `long double`
`FLT_RADIX`	Основание системы счисления (b), используемое в экспоненциальном представлении (минимальное значение — 2)
`FLT_MANT_DIG`	Количество цифр в представлении по основанию `FLT_RADIX` (p) в значащей части числа для `float`
`DBL_MANT_DIG`	Количество цифр в представлении по основанию `FLT_RADIX` (p) в значащей части числа для `double`
`LDBL_MANT_DIG`	Количество цифр в представлении по основанию `FLT_RADIX` (p) в значащей части числа для `long double`
`FLT_DECIMAL_DIG`	Количество десятичных цифр для `float`, которые могут быть преобразованы из основания b в основание 10 и обратно в основание b без изменения значения (минимальное значение — 6)

Продолжение табл. Б.V.9

Макрос	Описание
DBL_DECIMAL_DIG	Количество десятичных цифр для double, которые могут быть преобразованы из основания *b* в основание 10 и обратно в основание *b* без изменения значения (минимальное значение — 10)
LDBL_DECIMAL_DIG	Количество десятичных цифр для long double, которые могут быть преобразованы из основания *b* в основание 10 и обратно в основание *b* без изменения значения (минимальное значение — 10)
DECIMAL_DIG	Количество десятичных цифр для самого широкого поддерживаемого типа с плавающей запятой, которые могут быть преобразованы из основания *b* в основание 10 и обратно в основание *b* без изменения значения (минимальное значение — 10)
FLT_DIG	Количество десятичных цифр для float, которые могут быть преобразованы из основания 10 в основание *b* и обратно в основание 10 без изменения значения (минимальное значение — 6)
DBL_DIG	Количество десятичных цифр для double, которые могут быть преобразованы из основания 10 в основание *b* и обратно в основание 10 без изменения значения (минимальное значение — 10)
LDBL_DIG	Количество десятичных цифр для long double, которые могут быть преобразованы из основания 10 в основание *b* и обратно в основание 10 без изменения значения (минимальное значение — 10)
FLT_MIN_EXP	Минимальное отрицательное целочисленное значение, которое может принимать *е*, для float
DBL_MIN_EXP	Минимальное отрицательное целочисленное значение, которое может принимать *е*, для double
LDBL_MIN_EXP	Минимальное отрицательное целочисленное значение, которое может принимать *е*, для long double
FLT_MIN_10_EXP	Минимальное отрицательное целочисленное значение, такое что результат возведения 10 в степень, равную этому значению, по-прежнему является нормализованным числом float (не больше, чем -37)
DBL_MIN_10_EXP	Минимальное отрицательное целочисленное значение, такое что результат возведения 10 в степень, равную этому значению, по-прежнему является нормализованным числом double (не больше, чем -37)
LDBL_MIN_10_EXP	Минимальное отрицательное целочисленное значение, такое что результат возведения 10 в степень, равную этому значению, по-прежнему является нормализованным числом long double (не больше, чем -37)
FLT_MAX_EXP	Максимальное положительное целочисленное значение, которое может принимать *е*, для float
DBL_MAX_EXP	Максимальное положительное целочисленное значение, которое может принимать *е*, для double
LDBL_MAX_EXP	Максимальное положительное целочисленное значение, которое может принимать *е*, для long double
FLT_MAX_10_EXP	Максимальное положительное целочисленное значение, такое что результат возведения 10 в степень, равную этому значению, входит в диапазон представимых конечных значений float (по крайней мере +37)

Окончание табл. Б.V.9

Макрос	Описание
DBL_MAX_10_EXP	Максимальное положительное целочисленное значение, такое что результат возведения 10 в степень, равную этому значению, входит в диапазон представимых конечных значений double (по крайней мере +37)
LDBL_MAX_10_EXP	Максимальное положительное целочисленное значение, такое что результат возведения 10 в степень, равную этому значению, входит в диапазон представимых конечных значений long double (по крайней мере +37)
FLT_MAX	Максимальное представимое конечное значение float (по крайней мере, 1E+37)
DBL_MAX	Максимальное представимое конечное значение double (по крайней мере, 1E+37)
LDBL_MAX	Максимальное представимое конечное значение long double (по крайней мере, 1E+37)
FLT_EPSILON	Разность между 1 и наименьшим значением, большим 1, для float (не больше, чем 1E-5)
DBL_EPSILON	Разность между 1 и наименьшим значением, большим 1, для double (не больше, чем 1E-9)
LDBL_EPSILON	Разность между 1 и наименьшим значением, большим 1, для long double (не больше, чем 1E-9)
FLT_MIN	Наименьшее положительное нормализованное значение float (не больше, чем 1E-37)
DBL_MIN	Наименьшее положительное нормализованное значение double (не больше, чем 1E-37)
LDBL_MIN	Наименьшее положительное нормализованное значение long double (не больше, чем 1E-37)
FLT_TRUE_MIN	Наименьшее положительное значение float (не больше, чем 1E-37)
DBL_TRUE_MIN	Наименьшее положительное значение double (не больше, чем 1E-37)
LDBL_TRUE_MIN	Наименьшее положительное значение long double (не больше, чем 1E-37)

Преобразование формата целочисленных типов: inttypes.h (C99)

В заголовочном файле inttypes.h определено несколько макросов, которые могут использоваться в качестве спецификаторов формата для расширенных целочисленных типов. Более подробно это обсуждается в разделе VI приложения. В данном заголовочном файле также объявлен следующий тип:

```
imaxdiv_t
```

Этот тип представляет собой структуру, представляющую возвращаемое значение функции idivmax().

В inttypes.h также включен заголовочный файл stdint.h и объявлено несколько функций, которые работают с наиболее широким целочисленным типом, объявленным в stdint.h как intmax. Функции перечислены в табл. Б.V.10.

Таблица Б.V.10. Функции для работы с наиболее широким целочисленным типом

Прототип	Описание
intmax_t imaxabs(intmax_t j);	Возвращает абсолютное значение j
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);	Вычисляет частное и остаток от деления numer на denom в одной операции и сохраняет эти два значения в возвращаемой структуре
intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr, int base);	Эквивалент функции strtol() за исключением того, что строка преобразуется в тип intmax_t и это значение возвращается
uintmax_t strtoumax(const char * restrict nptr, char ** restrict endptr, int base);	Эквивалент функции strtoul() за исключением того, что строка преобразуется в тип uintmax_t и это значение возвращается
intmax_t wcstoimax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);	Версия wchar_t функции strtoimax()
uintmax_t wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);	Версия wchar_t функции strtoumax()

Альтернативное написание: iso646.h

Заголовочный файл iso646.h предоставляет 11 макросов, которые расширяются в указанные операции (табл. Б.V.11).

Таблица Б.V.11. Альтернативное написание

Макрос	Операция	Макрос	Операция	Макрос	Операция
and	&&	and_eq	&=	bitand	&
bitor	|	compl	~	not	!
not_eq	!=	or	||	or_eq	|=
xor	^	xor_eq	^=		

Локализация: locale.h

Локальная установка (или локаль) – это группа настроек, которые управляют такими элементами, как символ, используемый для представления десятичной точки. Локальные установки сохраняются в структуре типа struct lconv, которая определена в заголовочном файле locale.h. Локальная установка может быть задана строкой, которая указывает определенный набор значений для членов структуры. Стандартная локальная установка обозначается строкой "C". В табл. Б.V.12 перечислены функции локализации с кратким описанием каждой из них.

Таблица Б.V.12. Функции локализации

Прототип	Описание
`char * setlocale (int category, const char * locale);`	Эта функция устанавливает определенные значения локальной установки в значения, указанные в `locale`. Значение `category` управляет тем, какие именно значения локальной установки должны быть изменены (табл. Б.V.13). Функция возвращает нулевой указатель, если запрос не может быть удовлетворен. В противном случае она возвращает указатель, связанный с указанной категорией в новой локальной установке
`struct lconv * localeconv(void);`	Возвращает указатель на структуру `struct lconv`, заполненную значениями текущей локальной установки

Возможными значениями параметра `locale` при вызове `setlocale()` могут быть `"C"`, что принято по умолчанию, и `""`, что представляет собственную среду, определенную реализацией. Реализация может определять дополнительные локальные установки. Возможные значения параметра `category` при вызове `setlocale()` представлены макросами, которые перечислены в табл. Б.V.13.

Таблица Б.V.13. Макросы категорий

Макрос	Описание
`NULL`	Оставляет локальную установку неизмененной и возвращает указатель на текущую локальную установку
`LC_ALL`	Изменяет все локальные значения
`LC_COLLATE`	Изменяет локальные значения последовательности сопоставления, используемой `strcoll()` и `strxfrm()`
`LC_CTYPE`	Изменяет локальные значения, применяемые символьными функциями и многобайтными функциями
`LC_MONETARY`	Изменяет локальные значения, имеющие отношение к форматированию денежных величин
`LC_NUMERIC`	Изменяет локальные значения для символа десятичной точки и установки форматирования, не связанного с денежными значениями, которое используется при форматированном вводе-выводе и в функциях преобразования строк
`LC_TIME`	Изменяет локальные значения для форматирования показаний времени, используемых `strftime()`

В табл. Б.V.14 перечислены обязательные члены структуры `struct lconv`.

Таблица Б.V.14. Обязательные члены структуры `struct lconv`

Член	Описание
`char *decimal_point`	Символ десятичной точки для значений, отличных от денежных
`char *thousands_sep`	Символ, используемый для разделения групп цифр до точки для значений, отличных от денежных
`char *grouping`	Строка, чьи элементы указывают размер каждой группы цифр для значений, отличных от денежных
`char *int_curr_symbol`	Интернациональный символ валюты
`char *currency_symbol`	Локальный символ валюты
`char *mon_decimal_point`	Символ десятичной точки для денежных значений

Окончание табл. Б.V.14

Член	Описание
char *mon_thousands_sep	Символ, используемый для разделения групп цифр до точки для денежных значений
char *mon_grouping	Строка, элементы которой указывают размер каждой группы цифр для денежных значений
char *positive_sign	Строка, используемая для обозначения неотрицательных денежных значений
char *negative_sign	Строка, используемая для обозначения отрицательных денежных значений
char int_frac_digits	Количество цифр после точки для интернационально сформатированных денежных величин
char frac_digits	Количество цифр после точки для локально сформатированных денежных величин
char p_cs_precedes	Устанавливается в 1 или 0 в зависимости от того, предшествует currency_symbol или же следует за неотрицательными сформатированными денежными величинами
char p_sep_by_space	Устанавливается в 1 или 0 в зависимости от того, отделяется ли currency_symbol пробелом от неотрицательных сформатированных денежных величин
char n_cs_precedes	Устанавливается в 1 или 0 в зависимости от того, предшествует currency_symbol или же следует за отрицательными сформатированными денежными величинами
char n_sep_by_space	Устанавливается в 1 или 0 в зависимости от того, отделяется ли currency_symbol пробелом от отрицательных сформатированных денежных величин
char p_sign_posn	Устанавливает значение, указывающее позицию строки positive_sign; 0 означает, что скобки окружают число и символ валюты; 1 означает, что строка предшествует числу и символу валюты; 2 означает, что строка следует за числом и символом валюты; 3 означает, что строка предшествует непосредственно символу валюты; 4 означает, что строка следует немедленно за символом валюты
char n_sign_posn	Устанавливается в значение, указывающее положение строки negative_sign; принимает такие же значения, как p_sign_posn.
char int_p_cs_precedes	Устанавливается в 1 или 0 в зависимости от того, предшествует int_currency_symbol значению неотрицательных сформатированных денежных величин или следует за ним
char int_p_sep_by_space	Устанавливается в 1 или 0 в зависимости от того, отделяется ли пробелом int_currency_symbol от значения неотрицательных сформатированных денежных величин
char int_n_cs_precedes	Устанавливается в 1 или 0 в зависимости от того, предшествует int_currency_symbol значению отрицательных сформатированных денежных величин или следует за ним
char int_n_sep_by_space	Устанавливается в 1 или 0 в зависимости от того, отделяется ли пробелом int_currency_symbol от значения отрицательных сформатированных денежных величин
char int_p_sign_posn	Устанавливает значение, указывающее позицию positive_sign для неотрицательных интернационально сформатированных денежных величин
char int_n_sign_posn	Устанавливает значение, указывающее позицию negative_sign для неотрицательных интернационально сформатированных денежных величин

Математическая библиотека: math.h

В стандарте C99 внутри заголовочного файла math.h определены два типа:

```
float_t
double_t
```

Эти типы по ширине, по меньшей мере, соответствуют типам float и double, а double_t — по меньшей мере, типу float_t. Они предназначены для того, чтобы служить типами, обеспечивающими наиболее эффективные вычисления с данными float и double.

В math.h также определено несколько макросов, которые описаны в табл. Б.V.15; все они кроме HUGE_VAL добавлены стандартом C99. Некоторые из них более подробно обсуждаются в разделе VIII приложения.

Таблица Б.V.15. Макросы в math.h

Макрос	Описание
HUGE_VAL	Положительная константа double, не обязательно выражаемая как float; в прошлом она использовалась в качестве возвращаемого значения функций, когда абсолютная величина результата превышала максимальное представимое значение
HUGE_VALF	Дополнение HUGE_VAL типа float
HUGE_VALL	Дополнение HUGE_VAL типа long double
INFINITY	Расширяется до константного выражения float, представляющего положительную или беззнаковую бесконечность, если возможно; в противном случае расширяется до положительной константы float, которая приводит к переполнению на этапе компиляции
NAN	Определен тогда и только тогда, когда реализация поддерживает несигнальные NaN (значение, обозначающее не число) для типа float
FP_INFINITE	Классификационное число, символизирующее бесконечное значение с плавающей запятой
FP_NAN	Классификационное число, символизирующее значение с плавающей запятой, которое не является числом
FP_NORMAL	Классификационное число, символизирующее нормальное значение с плавающей запятой
FP_SUBNORMAL	Классификационное число, символизирующее субнормальное значение с плавающей запятой (с пониженной точностью)
FP_ZERO	Классификационное число, символизирующее значение с плавающей запятой, которое представляет 0
FP_FAST_FMA	(Необязательный.) Если определен, этот макрос говорит о том, что функция fma() работает почти так же быстро или быстрее, чем умножение и сложение операндов типа double
FP_FAST_FMAF	(Необязательный.) Если определен, этот макрос говорит о том, что функция fmaf() работает почти так же быстро или быстрее, чем умножение и сложение операндов типа float
FP_FAST_FMAL	(Необязательный.) Если определен, этот макрос говорит о том, что функция fmal() работает почти так же быстро или быстрее, чем умножение и сложение операндов типа long double

Окончание табл. Б.V.15

Макрос	Описание
FP_ILOGB0	Целое константное выражение, которое представляет значение, возвращаемое ilogb(0)
FP_ILOGBNAN	Целое константное выражение, которое представляет значение, возвращаемое ilogb(NaN)
MATH_ERRNO	Расширяется до целочисленной константы 1
MATH_ERREXCEPT	Расширяется до целочисленной константы 2
math_errhandling	Имеет значение MATH_ERRNO или MATH_ERREXCEPT либо объединение этих двух значений с помощью побитового "ИЛИ"

Математические функции обычно работают со значениями типа double. В стандарте C99 были добавлены версии float и long double этих функций, что отмечается дополнением их имен соответственно суффиксами f и l. Например, теперь доступны следующие прототипы:

```
double sin(double);
float sinf(float);
long double sinl(long double);
```

Для краткости в табл. Б.V.16 перечислены только версии double функций математической библиотеки. В таблице присутствует ссылка на константу FLT_RADIX. Эта константа, определенная в float.h, в основном используется для возведения в степень во внутреннем представлении значения плавающей запятой. Чаще всего она равна 2.

Таблица Б.V.16. Стандартные математические функции ANSI C

Прототип	Описание
int classify(реальный-тип-с-плавающей-запятой x);	Макрос C99, который возвращает классификационное значение плавающей запятой, соответствующее x
int isfinite(реальный-тип-с-плавающей-запятой x);	Макрос C99, который возвращает ненулевое значение тогда и только тогда, когда x конечно
int isfin(реальный-тип-с-плавающей-запятой x);	Макрос C99, который возвращает ненулевое значение тогда и только тогда, когда x бесконечно
int isnan(реальный-тип-с-плавающей-запятой x);	Макрос C99, который возвращает ненулевое значение тогда и только тогда, когда x — NaN
int isnormal(реальный-тип-с-плавающей-запятой x);	Макрос C99, который возвращает ненулевое значение тогда и только тогда, когда x нормально
int signbit(реальный-тип-с-плавающей-запятой x);	Макрос C99, который возвращает ненулевое значение тогда и только тогда, когда x отрицательно
double acos(double x);	Возвращает угол (от 0 до π радиан), косинус которого равен x
double asin(double x);	Возвращает угол (от -π /2 до π /2 радиан), синус которого равен x
double atan(double x);	Возвращает угол (от -π/2 до π/2 радиан), тангенс которого равен x
double atan2(double y, double x);	Возвращает угол (от -π до π радиан), тангенс которого равен x/y
double cos(double x);	Возвращает косинус x (x в радианах)
double sin(double x);	Возвращает синус x (x в радианах)

Продолжение табл. Б.V.16

Прототип	Описание
`double tan(double x);`	Возвращает тангенс x (x в радианах)
`double cosh(double x);`	Возвращает гиперболический косинус x
`double sinh(double x);`	Возвращает гиперболический синус x
`double tanh(double x);`	Возвращает гиперболический тангенс x
`double exp(double x);`	Возвращает экспоненциальную функцию x (e^x)
`double exp2(double x);`	Возвращает 2 в степени x (C99)
`double expm1(double x);`	Возвращает $e^x - 1$ (C99)
`double frexp(double v, int *pt_e);`	Разбивает значение v на нормализованную дробную часть, которая возвращается, и степень 2, которая помещается в место, указанное pt_e
`int ilogb(double x);`	Возвращает экспоненту x как целое со знаком (C99)
`double ldexp(double x, int p);`	Возвращает 2 в степени p, умноженное на x
`double log(double x);`	Возвращает натуральный логарифм x
`double log10(double x);`	Возвращает десятичный логарифм x
`double log1p(double x);`	Возвращает `log(1 + x)` (C99)
`double log2(double x);`	Возвращает двоичный логарифм x
`double logb(double x);`	Возвращает экспоненту аргумента со знаком для базового основания системы счисления, используемого в системе при представлении значений с плавающей запятой (FLT_RADIX) (C99)
`double modf(double x, double *p);`	Разбивает значение x на целую и дробную части, обе с одинаковым знаком, возвращает дробную часть, а целую помещает в место, указанное p
`double scalbn(double x, int n);`	Возвращает x × FLT_RADIX^n (C99)
`double scalbn(double x, long n);`	Возвращает x × FLT_RADIX^n (C99)
`double cbrt(double x);`	Возвращает кубический корень из x (C99)
`double hypot(double x, double y);`	Возвращает квадратный корень из суммы квадратов x и y (C99)
`double pow(double x, double y);`	Возвращает x в степени y
`double sqrt(double x);`	Возвращает квадратный корень из x
`double erf(double x);`	Возвращает функцию ошибки x
`double erfc(double x);`	Возвращает дополнительную функцию ошибки x
`double lgamma(double x);`	Возвращает натуральный логарифм абсолютного значения гамма-функции x (C99)
`double tgamma(double x);`	Возвращает гамма-функцию x (C99)
`double ceil(double x);`	Возвращает минимальное целое значение, не меньшее чем x
`double fabs(double x);`	Возвращает абсолютное значение x

Продолжение табл. Б.V.16

Прототип	Описание
`double floor(double x);`	Возвращает максимальное целое значение, не большее чем x
`double nearbyint(double x);`	Округляет x до ближайшего целого в формате с плавающей запятой; использует направление округления, установленное средой плавающей запятой, если она доступна. Исключение "неточности" не генерируется (С99)
`double rint(double x);`	Подобна `nearbyint()`, но может генерировать исключение "неточности" (С99)
`long int lrint(double x);`	Округляет x до ближайшего целого в формате `long int`; использует направление округления, установленное средой плавающей запятой, если она доступна (С99)
`long long int llrint( double x);`	Округляет x до ближайшего целого в формате `long long int`; использует направление округления, установленное средой плавающей запятой, если она доступна (С99)
`double round(double x);`	Округляет x до ближайшего целого в формате с плавающей запятой; всегда округляет частичные значения в сторону от нуля (С99)
`long int lround(double x);`	Подобна `round()`, но возвращает ответ типа `long int` (С99)
`long long int lround( double x);`	Подобна `round()`, но возвращает ответ типа `long long int` (С99)
`double trunc(double x);`	Округляет x до ближайшего целого в формате с плавающей запятой, которое не больше абсолютного значения x (С99)
`int fmod(double x, double y);`	Возвращает дробную часть `x/y`; если y — не ноль, то результат получает тот же знак, что x, и по абсолютному значению меньше, чем y
`double remainder(double x, double y);`	Возвращает x `REM` y, что в стандарте IEC 60559 определено как `x - n*y`, где n — ближайшее к `x/y` целое; n — четное, если абсолютное значение `(n — x/y)` равно 1/2 (С99)
`double remquo(double x, double y, int *quo);`	Возвращает то же значение, что и `reminder()`, и помещает в место, указываемое `quo`, значение, имеющее тот же знак, что и `x/y`, и имеющее абсолютную целую величину `x/y` по модулю 2^k, где k — зависящее от реализации целое, значение которого не меньше 3 (С99)
`double copysign(double x, double y);`	Возвращает значение абсолютной величины x со знаком y (С99)
`double nan( const char *tagp);`	Возвращает `double`-представление `NaN`; `nan("`*последовательность-символов-n*`")` эквивалентно `strtod("NAN(`*последовательность-символов-n*`)", (char **) NULL)`; `nan("")` — эквивалент `strtod("NAN()", (char **) NULL)`; для других строк, передаваемых в качестве аргументов, вызов эквивалентен `strtod("NAN", (char **)NULL)`. Возвращает 0, если несигнальные `NaN` не поддерживаются (С99)
`double nextafter(double x, double y);`	Возвращает следующее представимое значение типа `double` после x в направлении y; возвращает x, если x равно y (С99)
`double nexttoward(double x, long double y);`	То же самое, что и `nextafter()`, за исключением того, что второй аргумент имеет тип `long double`, и если x равно y, функция возвращает y, преобразованное в `double` (С99)

Окончание табл. Б.V.16

Прототип	Описание
`double fdim(double x, double y);`	Возвращает положительную разность аргументов (C99)
`double fmax(double x, double y);`	Возвращает максимальное числовое значение из двух аргументов; если один из аргументов NaN, а второй — число, возвращается второй аргумент (C99)
`double fmin(double x, double y);`	Возвращает минимальное числовое значение из двух аргументов; если один из аргументов NaN, а второй — число, возвращается второй аргумент (C99)
`double fma(double x, double y, double z);`	Возвращает величину (x*y)+z как тернарную операцию, округленную один раз в конце (C99)
`int isgreater(реальный-тип-с-плавающей-запятой x, реальный-тип-с-плавающей-запятой y);`	Макрос C99, который возвращает значение (x) > (y) без генерации исключения плавающей запятой типа "некорректное число", если один или оба аргумента являются NaN
`int isgreaterequal(реальный-тип-с-плавающей-запятой x, реальный-тип-с-плавающей-запятой y);`	Макрос C99, который возвращает значение (x) >= (y) без генерации исключения плавающей запятой типа "некорректное число", если один или оба аргумента являются NaN
`int isless(реальный-тип-с-плавающей-запятой x, реальный-тип-с-плавающей-запятой y);`	Макрос C99, который возвращает значение (x) < (y) без генерации исключения плавающей запятой типа "некорректное число", если один или оба аргумента являются NaN
`int islessequal(реальный-тип-с-плавающей-запятой x, реальный-тип-с-плавающей-запятой y);`	Макрос C99, который возвращает значение (x) <= (y) без генерации исключения плавающей запятой типа "некорректное число", если один или оба аргумента являются NaN
`int islessgreater(реальный-тип-с-плавающей-запятой x, реальный-тип-с-плавающей-запятой y);`	Макрос C99, который возвращает значение (x) < (y) || (x) > (y) без генерации исключения плавающей запятой типа "некорректное число", если один или оба аргумента являются NaN
`int isunordered(реальный-тип-с-плавающей-запятой x, реальный-тип-с-плавающей-запятой y);`	Возвращает единицу, если аргументы неупорядочены (т.е. хотя бы один является NaN), в противном случае возвращает 0

Нелокальные переходы: setjmp.h

Заголовочный файл setjmp.h позволяет обходить обычную последовательность вызовов и возвращений из функций. Функция setjmp() сохраняет информацию о текущей среде выполнения (например, указатель на текущую инструкцию) в переменной типа jmp_buf (тип массива, определенный в setjmp.h), а функция longjmp() передает выполнение этой среде. Функции предназначены для помощи в обработке ошибочных ситуаций и не задуманы для использования как части нормального потока управления программы. Упомянутые функции описаны в табл. Б.V.17.

Таблица Б.V.17. Функции в setjmp.h

Прототип	Описание
`int setjmp(jmp_buf env);`	Сохраняет текущую среду вызовов в массиве `env` и возвращает `0`, если она вызвана напрямую, и `1`, если произошел возврат из вызова `longjmp()`
`void longjmp(jmp_buf env, int val);`	Восстанавливает среду, сохраненную последним вызовом `setjmp()`, который записал ее в массив `env`; после этого программа продолжает выполнение, как будто предыдущий вызов `setjmp()` возвратил `val`, но возвращаемое значение `1` не разрешено и было преобразовано в `0`

Обработка сигналов: signal.h

Сигнал – это условие, которое может быть сообщено программе во время ее выполнения. Он представляется положительным целым числом. Функция `raise()` отправляет, или *генерирует*, сигнал, а функция `signal()` устанавливает ответ на определенный сигнал.

В стандарте определен целочисленный тип, `sig_atomic_t`, используемый для указания объектов, которые являются атомарными в отношении обработчиков сигналов. Другими словами, обновление атомарного типа представляет собой неделимый процесс.

Стандарт предоставляет макросы, перечисленные в табл. Б.V.18, которые предназначены для представления возможных сигналов; реализация может добавлять дополнительные значения. Эти макросы могут использоваться в качестве аргументов функций `raise()` и `signal()`.

Таблица Б.V.18. Макросы сигналов

Макрос	Описание
SIGABRT	Ненормальное завершение, такое как инициированное вызовом `abort()`
SIGFPE	Ошибочная арифметическая операция
SIGILL	Обнаружен некорректный образ функции (такой как недопустимая инструкция)
SIGINT	Принят интерактивный сигнал внимания (такой как прерывание DOS)
SIGSERV	Некорректное обращение к хранилищу
SIGTERM	Программа получила запрос на прекращение работы

В качестве второго аргумента функция `signal()` принимает указатель на функцию `void`, получающую аргумент `int`. Она также возвращает указатель того же типа. Функция, вызываемая в ответ на сигнал, называется *обработчиком сигнала*. Стандарт определяет три макроса, подходящие этому прототипу:

```
void (*funct)(int);
```

В табл. Б.V.19 перечислены эти макросы.

Если сигнал `sig` сгенерирован, а `func` указывает на функцию (см. прототип `signal()` в табл. Б.V.20), то сначала в большинстве случаев вызывается `signal(sig, SIG_DFL)` для сброса обработчика сигнала к стандартной установке, после чего вызывается `(*func)(sig)`.

Таблица Б.V.19. Макросы типа void (*f) (int)

Макрос	Описание
SIG_DFL	Когда используется в качестве аргумента signal() наряду со значением сигнала, этот макрос указывает, что произойдет стандартная обработка для данного сигнала
SIG_ERR	Применяется в качестве возвращаемого значения signal(), если она не может возвратить свой второй аргумент
SIG_IGN	Когда используется в качестве аргумента signal() наряду со значением сигнала, этот макрос указывает, что сигнал будет проигнорирован

Функция обработки сигнала, указанная с помощью func, может быть завершена выполнением оператора return либо вызовом abort(), exit() или longjmp().

В табл. Б.V.20 перечислены функции сигналов.

Таблица Б.V.20. Функции сигналов

Прототип	Описание
void (*signal(int sig, void(*func)(int)))(int);	Заставляет функцию, на которую указывает func, вызываться, если сгенерирован сигнал sig. По возможности возвращает func, иначе — SIG_ERR
int raise(int sig);	Отправляет сигнал sig выполняющейся программе; в случае успеха возвращает ноль и ненулевое значение в противном случае

Выравнивание: stdalign.h (C11)

В заголовочном файле stdalign.h определены четыре макроса, имеющие отношение к определению и указанию свойств выравнивания объектов данных. Эти макросы перечислены в табл. Б.V.21. Первые два макроса создают псевдонимы, совместимые с C++.

Таблица Б.V.21. Макросы в stdalign.h

Макрос	Описание
alignas	Расширяется до ключевого слова _Alignas
alignof	Расширяется до ключевого слова _Alignof
__alignas_is_defined	Расширяется до целочисленной константы 1, подходящей для использования с #if
__alignof_is_defined	Расширяется целочисленной константы 1, подходящей для использования с #if

Переменное количество аргументов: stdarg.h

Заголовочный файл stdarg.h предоставляет средства для определения функций, принимающих переменное количество аргументов. Прототип для такой функции должен содержать список параметров, в котором указан как минимум один параметр, за которым следует троеточие:

```
void f1(int n, ...);                    /* допустимо   */
int f2(int n, float x, int k, ...);     /* допустимо   */
double  f3(...);                        /* недопустимо */
```

В следующей таблице термин *parmN* — это идентификатор, используемый для обозначения последнего параметра, который предшествует троеточию. В предыдущих примерах таким параметром был n в первом случае и k — во втором.

В заголовочном файле объявлен тип va_list для представления объекта данных, который применяется для хранения параметров, соответствующих троеточию в списке параметров. В табл. Б.V.22 перечислены макросы, которые должны использоваться в функциях с переменным количеством параметров. Перед применением этих макросов должен быть объявлен объект типа va_list.

Таблица Б.V.22. Макросы переменных списков аргументов

Прототип	Описание
`void va_start(va_list ap, parmN);`	Этот макрос инициализирует ap перед использованием va_arg() и va_end(); *parmN* — идентификатор последнего именованного параметра в списке аргументов
`void va_copy(va_list dest, va_list src);`	Этот макрос инициализирует dest как копию текущего состояния src (C99)
`type va_arg(va_list ap, type);`	Этот макрос расширяется до выражения, имеющего то же значение и тип, что и следующий элемент в списке аргументов, представленном ap; *type* — тип этого элемента. Каждый вызов переходит к следующему элементу ap
`void va_end(va_list ap);`	Этот макрос прекращает процесс и может сделать ap недоступным без повторного вызова va_start()

Поддержка атомарности: stdatomic.h (C11)

Заголовочный файл stdatomic.h вместе с threads.h обеспечивает поддержку для параллельного программирования. Эта тема выходит за рамки настоящей книги, но если выражаться общими терминами, то заголовочный файл stdatomic.h предоставляет макросы для создания атомарных операций. В сообществе программистов понятие *атомарный* используется в том же смысле, в каком Демокрит применял его в своей теории материи — т.е. неделимый. Операция, такая как присваивание одной структуры другой, на уровне программирования может выглядеть атомарной, но на уровне машинного языка состоять из нескольких шагов. Если программа разделена на множество потоков, то один поток может читать или модифицировать данные, которые находятся в процессе использования другим потоком. В итоге могла бы получиться странная структура, в которой значения одних членов установлены одним потоком, а других членов — другим потоком. Заголовочный файл stdatomic.h позволяет создавать операции, которые действуют так, как если бы они были атомарными, т.е. один поток не может прерывать работу другого потока.

Поддержка булевских значений: stdbool.h (C99)

В этом заголовочном файле определены четыре макроса, описанные в табл. Б.V.23.

Таблица Б.V.23. Макросы в stdbool.h

Макрос	Описание
bool	Расширяется до _Bool
false	Расширяется до целочисленной константы 0
true	Расширяется до целочисленной константы 1
__bool_true_false_are_defined	Расширяется до целочисленной константы 1

Общие определения: stddef.h

В этом заголовочном файле определен ряд типов и макросов, которые показаны в табл. Б.V.24 и Б.V.25.

Таблица Б.V.24. Типы в stddef.h

Тип	Описание
ptrdiff_t	Целочисленный тип со знаком для представления результата вычитания одного указателя из другого
size_t	Целочисленный тип без знака для представления результата операции sizeof
wchar_t	Целочисленный тип, который может представлять наибольший расширенный набор символов, указанный поддерживаемыми локальными установками

Таблица Б.V.25. Макросы в stddef.h

Тип	Описание
NULL	Определяемая реализацией константа, представляющая нулевой указатель
offsetof(*тип*, *указатель-члена*)	Расширяется до значения size_t, представляющего смещение в байтах указанного члена от начала структуры типа *тип*; поведение не определено, если член является битовым полем

Пример

```
#include <stddef.h>
struct car
{
   char brand[30];
   char model[30];
   double hp;
   double price;
};
int main(void)
{
   size_t into = offsetof(struct car, hp);  /* смещение члена hp */
   ...
```

Целочисленные типы: stdint.h

В этом заголовочном файле с помощью средства typedef создаются имена целочисленных типов, которые указывают свойства целых чисел. Файл stdint.h включен в заголовочный файл inttypes.h, который предоставляет макросы для использования в вызовах функций ввода-вывода. Возможные применения этих типов кратко описаны в разделе VI данного приложения.

Типы с точной шириной

Один из наборов typedef идентифицирует типы с точными размерами. Их имена и размеры перечислены в табл. Б.V.26. Однако обратите внимание, что не все системы могут поддерживать все эти типы.

Таблица Б.V.26. Типы с точной шириной

Имя *typedef*	Описание
`int8_t`	8-битовый со знаком
`int16_t`	16-битовый со знаком
`int32_t`	32-битовый со знаком
`int64_t`	64-битовый со знаком
`uint8_t`	8-битовый без знака
`uint16_t`	16-битовый без знака
`uint32_t`	32-битовый без знака
`uint64_t`	64-битовый без знака

Типы с минимальной шириной

Типы с минимальной шириной гарантируют, что тип имеет размер, равный, как минимум, определенному количеству битов. Типы с минимальной шириной перечислены в табл. Б.V.27. Такие типы существуют всегда.

Таблица Б.V.27. Типы с минимальной шириной

Имя *typedef*	Описание
`int_least8_t`	По крайней мере, 8 битов со знаком
`int_least16_t`	По крайней мере, 16 битов со знаком
`int_least32_t`	По крайней мере, 32 бита со знаком
`int_least64_t`	По крайней мере, 64 бита со знаком
`uint_least8_t`	По крайней мере, 8 битов без знака
`uint_least16_t`	По крайней мере, 16 битов без знака
`uint_least32_t`	По крайней мере, 32 бита без знака
`uint_least64_t`	По крайней мере, 64 бита без знака

Самые быстрые типы с минимальной шириной

В отдельной системе некоторые представления целых чисел могут быть быстрее других. Поэтому в `stdint.h` также определены самые быстрые типы для представления, по крайней мере, определенного количества битов. В табл. Б.V.28 перечислены наиболее быстрые типы с минимальной шириной. Такие типы также существуют всегда. В некоторых случаях может отсутствовать очевидный выбор для самого быстрого типа, тогда система просто указывает один из возможных вариантов.

Таблица Б.V.28. Самые быстрые типы с минимальной шириной

Имя *typedef*	Описание
`int_fast8_t`	По крайней мере, 8 битов со знаком
`int_fast16_t`	По крайней мере, 16 битов со знаком
`int_fast32_t`	По крайней мере, 32 бита со знаком
`int_fast64_t`	По крайней мере, 64 бита со знаком
`uint_fast8_t`	По крайней мере, 8 битов без знака
`uint_fast16_t`	По крайней мере, 16 битов без знака
`uint_fast32_t`	По крайней мере, 32 бита без знака
`uint_fast64_t`	По крайней мере, 64 бита без знака

Типы с максимальной шириной

В заголовочном файле stdint.h также определены типы с максимальной шириной. Переменная такого типа может содержать любое целочисленное значение, возможное для системы, с учетом знака. Эти типы перечислены в табл. Б.V.29.

Таблица Б.V.29. Типы с максимальной шириной

Имя *typedef*	Описание
intmax_t	Самый широкий целочисленный тип со знаком
uintmax_t	Самый широкий целочисленный тип без знака

Целые числа, которые могут хранить значения указателей

В заголовочном файле stdint.h также определены два целочисленных типа (табл. Б.V.30), которые могут точно хранить значения указателей. Другими словами, если переменной одного из таких типов присвоить значение типа void *, а затем присвоить значение этой переменной обратно указателю, то информация не теряется. В конкретной реализации может отсутствовать любой из этих типов или же оба.

Таблица Б.V.30. Целочисленные типы для хранения значений указателей

Имя *typedef*	Описание
intptr_t	Целочисленный тип со знаком, способный хранить указатели
uintptr_t	Целочисленный тип без знака, способный хранить указатели

Определенные константы

В заголовочном файле stdint.h также определены константы, представляющие предельные значения для типов, которые определены в этом файле. Константы названы по именам типов. Чтобы получить имя константы, представляющей минимальное или максимальное значение данного типа, возьмите имя типа, замените _t на _MAX или _MIN и переведите все символы в верхний регистр. Например, наименьшим значением для типа int32_t является INT32_MIN, а наибольшим значением для типа uint_fast16_t — UINT_FAST16_MAX. В табл. Б.V.31 приведена сводка по эти константам (*N* обозначает количество битов), а также по константам, относящимся к типам intptr_t, uintptr_t, intmax_t и intmax_t. Величины этих констант будут равны или превышать (если только не указано "в точности") перечисленные значения.

Таблица Б.V.31. Целочисленные константы

Идентификатор константы	Минимальное значение
INT*N*_MIN	В точности $-(2^{N-1} - 1)$
INT*N*_MAX	В точности $2^{N-1} - 1$
UINT*N*_MAX	В точности $2^{N} - 1$
INT_LEAST*N*_MIN	$-(2^{N-1} - 1)$
INT_LEAST*N*_MAX	$2^{N-1} - 1$
UINT_LEAST*N*_MAX	$2^{N} - 1$
INT_FAST*N*_MIN	$-(2^{N-1} - 1)$

Окончание табл. Б.V.16

Идентификатор константы	Минимальное значение
INT_FASTN_MAX	$2^{N-1} - 1$
UINT_FASTN_MAX	$2^{N} - 1$
INTPTR_MIN	$-(2^{15} - 1)$
INTPTR_MAX	$2^{15} - 1$
UINTPTR_MAX	$2^{16} - 1$
INTMAX_MIN	$-(2^{15} - 1)$
INTMAX_MAX	$2^{63} - 1$
UINTMAX_MAX	$2^{64} - 1$

В этом заголовочном файле также определены некоторые константы для типов, определенных где-то в других местах. Они перечислены в табл. Б.V.32.

Таблица Б.V.32. Дополнительные целочисленные константы

Идентификатор константы	Описание
PTRDIFF_MIN	Минимальное значение типа ptrdiff_t
PTRDIFF_MAX	Максимальное значение типа ptrdiff_t
SIG_ATOMIC_MIN	Минимальное значение типа sig_atomic_t
SIG_ATOMIC_MAX	Максимальное значение типа sig_atomic_t
WCHAR_MIN	Минимальное значение типа wchar_t
WCHAR_MAX	Максимальное значение типа wchar_t
WINT_MIN	Минимальное значение типа wint_t
WINT_MAX	Максимальное значение типа wint_t
SIZE_MAX	Максимальное значение типа size_t

Расширенные целочисленные константы

В заголовочном файле stdint.h определены макросы для указания констант разнообразных расширенных целочисленных типов. По существу такой макрос является приведением к лежащему в основе типу, т.е. к фундаментальному типу, который представляет расширенный тип в конкретной реализации.

Для формирования имени макроса возьмите имя типа, замените _t на _C и переведите все буквы в верхний регистр. Например, чтобы сделать 1000 константой типа uint_least64_t, используйте выражение UINT_LEAST64_C(1000).

Стандартная библиотека ввода-вывода: stdio.h

Стандартная библиотека ANSI С содержит множество стандартных функций ввода-вывода, ассоциированных с потоками и файлом stdio.h. В табл. Б.V.33 представлены прототипы ANSI для этих функций вместе с кратким объяснением их работы. (Многие функции были более подробно описаны в главе 13.) Кроме того, в заголовочном файле stdio.h определен тип FILE, значения EOF и NULL, а также стандартные потоки ввода-вывода stdin, stdout и stderr, наряду с константами, которые используются функциями в этой библиотеке.

Таблица Б.V.33. Стандартные функции ввода-вывода С

Прототип	Описание
`void clearerr(FILE *);`	Очищает индикаторы конца файла и ошибки
`int fclose(FILE *);`	Закрывает указанный файл
`int feof(FILE *);`	Проверяет, достигнут ли конец файла
`int ferror(FILE *);`	Проверяет индикатор ошибки
`int fflush(FILE *);`	Сбрасывает буфер указанного файла
`int fgetc(FILE *);`	Получает следующий символ из указанного входного потока
`int fgetpos(FILE * restrict, fpos_t * restrict);`	Сохраняет текущее значение индикатора позиции файла
`char * fgets(char * restrict, FILE * restrict);`	Получает следующую строку (или значение `int`, отражающее количество символов) из указанного потока
`FILE * fopen(const char * restrict, const char *restrict);`	Открывает указанный файл
`int fprintf(FILE * restrict, const char * restrict, ...);`	Записывает форматированный вывод в указанный поток
`int fputc(int, FILE *);`	Записывает в указанный поток заданный символ
`int fputs(const char * restrict, FILE * restrict);`	Записывает в указанный поток символьную строку, на которую указывает первый аргумент
`size_t fread(void * restrict, size_t, size_t, FILE * restrict);`	Читает двоичные данные из указанного потока
`FILE * freopen(const char * restrict, const char * restrict, FILE * restrict);`	Открывает заданный файл и ассоциирует его с указанным потоком
`int fscanf(FILE * restrict, const char * restrict, ...);`	Читает форматированный ввод из указанного потока
`int fsetpos(FILE *,const fpos_t *);`	Устанавливает указатель позиции файла в заданное значение
`int fseek(FILE *, long, int);`	Перемещает указатель позиции файла в заданное положение
`long ftell(FILE *);`	Получает текущую позицию файла
`size_t fwrite(const void * restrict, size_t, size_t, FILE * restrict);`	Записывает двоичные данные в указанный поток
`int getc(FILE *);`	Читает следующий символ из указанного ввода
`int getchar();`	Читает следующий символ из стандартного ввода
`char * gets(char *);`	Получает следующую строку из стандартного ввода (удалена из библиотеки стандартом C11)
`void perror(const char *);`	Записывает системные сообщения об ошибках в стандартный поток ошибок
`int printf(const char * restrict, ...);`	Записывает форматированный вывод в стандартный вывод
`int putc(int, FILE *);`	Записывает заданный символ в указанный вывод
`int putchar(int);`	Записывает заданный символ в стандартный вывод

Окончание табл. Б.V.33

Прототип	Описание
`int puts(const char *);`	Записывает строку в стандартный вывод
`int remove(const char *);`	Удаляет файл с указанным именем
`int rename(const char *, constchar *);`	Переименовывает файл с указанным именем
`void rewind(FILE *);`	Устанавливает указатель позиции в начало файла
`int scanf(const char * restrict, ...);`	Читает форматированный ввод из стандартного ввода
`void setbuf(FILE * restrict, char * restrict);`	Устанавливает размер и местоположение буфера
`int setvbuf(FILE * restrict, char *restrict, int, size_t);`	Устанавливает размер, местоположение и режим буфера
`int snprintf(char * restrict, size_t n, const char * restrict, ...);`	Записывает до `n` символов форматированного вывода в указанную строку
`int sprintf(char * restrict, const char * restrict, ...);`	Записывает форматированный вывод в указанную строку
`int sscanf(const char *restrict, const char * restrict, ...);`	Читает форматированный ввод из указанной строки
`FILE * tmpfile(void);`	Создает временный файл
`char * tmpnam(char *);`	Генерирует уникальное имя для временного файла
`int ungetc(int, FILE *);`	Заталкивает указанный символ обратно во входной поток
`int vfprintf(FILE * restrict, const char * restrict, va_list);`	Подобна `fprintf()`, но вместо списка аргументов переменной длины использует единственный списковый аргумент типа `va_list`, инициализированный `va_start`
`int vprintf(const char * restrict, va_list);`	Подобна `printf()`, но вместо списка аргументов переменной длины использует единственный списковый аргумент типа `va_list`, инициализированный `va_start`
`int vsnprintf(char * restrict, size_t n, const char * restrict, va_list);`	Подобна `snprintf()`, но вместо списка аргументов переменной длины использует единственный списковый аргумент типа `va_list`, инициализированный `va_start`
`int vsprintf(char * restrict, const char * restrict, va_list);`	Подобна `sprintf()`, но вместо списка аргументов переменной длины использует единственный списковый аргумент типа `va_list`, инициализированный `va_start`
`int vscanf(const char * restrict, va list);`	Подобна `scanf()`, но вместо списка аргументов переменной длины использует единственный списковый аргумент типа `va_list`, инициализированный `va_start`
`int vsscanf(const char * restrict, const char * restrict, va_list);`	Подобна `sscanf()`, но вместо списка аргументов переменной длины использует единственный списковый аргумент типа `va_list`, инициализированный `va_start`

Общие утилиты: stdlib.h

Стандартная библиотека ANSI C включает множество служебных функций, определенных в stdlib.h. В этом заголовочном файле определены типы, перечисленные в табл. Б.V.34.

Таблица Б.V.34. Типы, объявленные в stdlib.h

Тип	Описание
size_t	Целочисленный тип, возвращаемый операцией sizeof
wchar_t	Целочисленный тип, используемый для представления широких символов
div_t	Тип структуры, возвращаемый div(); содержит члены quot и rem, оба типа int
ldiv_t	Тип структуры, возвращаемый ldiv(); содержит члены quot и rem, оба типа long
lldiv_t	Тип структуры, возвращаемый lldiv(); содержит члены quot и rem, оба типа long long (C99)

В заголовочном файле также определены константы, описанные в табл. Б.V.35.

Таблица Б.V.35. Константы, определенные в stdlib.h

Тип	Описание
NULL	Нулевой указатель (эквивалент 0)
EXIT_FAILURE	Может использоваться в качестве аргумента функции exit() для обозначения неудачного завершения программы
EXIT_SUCCESS	Может использоваться в качестве аргумента функции exit() для обозначения успешного завершения программы
RAND_MAX	Максимальное значение (целое число), возвращаемое функцией rand()
MB_CUR_MAX	Максимальное количество байтов в многобайтном символе из расширенного набора символов, соответствующего текущей локальной установке

В табл. Б.V.36 представлены прототипы функций, определенные в stdlib.h.

Таблица Б.V.36. Утилиты общего назначения

Прототип	Описание
double atof(const char * nptr);	Возвращает начальную часть строки nptr, преобразованную в значение типа double; преобразование завершается при обнаружении первого символа, не являющегося частью числа; начальные пробелы пропускаются; если число в строке не найдено, возвращается ноль
int atoi(const char * nptr);	Возвращает начальную часть строки nptr, преобразованную в значение типа int; преобразование завершается при обнаружении первого символа, не являющегося частью числа; начальные пробелы пропускаются; если число в строке не найдено, возвращается ноль
long atol(const char * nptr);	Возвращает начальную часть строки nptr, преобразованную в значение типа long; преобразование завершается при обнаружении первого символа, не являющегося частью числа; начальные пробелы пропускаются; если число в строке не найдено, возвращается ноль

Продолжение табл. Б.V.36

Прототип	Описание
`double strtod( char * restrict npt, char ** restrict ept);`	Возвращает начальную часть строки `npt`, преобразованную в значение типа `double`; преобразование завершается при обнаружении первого символа, не являющегося частью числа; начальные пробелы пропускаются; если число в строке не найдено, возвращается ноль. Если преобразование прошло успешно, адрес первого символа после числа сохраняется по адресу, указанному `ept`, а если неудачно — `npt` записывается по адресу, указанному `ept`
`float strtof( const char * restrict npt, char ** restrict ept);`	То же, что и `strtod()`, но преобразует строку, на которую указывает `npt`, в значение типа `float` (C99)
`long double strtols( const char * restrict npt, char ** restrict ept);`	То же, что и `strtod()`, но преобразует строку, на которую указывает `npt`, в значение типа `long double` (C99)
`long strtol( const char * restrict npt, char ** restrict ept, int base);`	Возвращает начальную часть строки `npt`, преобразованную в значение типа `long`; преобразование завершается при обнаружении первого символа, не являющегося частью числа; начальные пробелы пропускаются; если число в строке не найдено, возвращается ноль. Если преобразование прошло успешно, адрес первого символа после числа сохраняется по адресу, указанному `ept`, а если неудачно — `npt` записывается по адресу, указанному `ept`. Предполагается, что число в строке записано с основанием, заданным в `base`
`long long strtoll( const char * restrict npt, char ** restrict ept, int base);`	То же, что и `strtol()`, но преобразует строку, указанную `npt`, в значение типа `long long` (C99)
`unsigned long strtol( const char * restrict npt, char ** restrict ept, int base);`	Возвращает начальную часть строки `npt`, преобразованную в значение типа `unsigned long`; преобразование завершается при обнаружении первого символа, не являющегося частью числа; начальные пробелы пропускаются; если число в строке не найдено, возвращается ноль. Если преобразование прошло успешно, адрес первого символа после числа сохраняется по адресу, указанному `ept`, а если неудачно — `npt` записывается по адресу, указанному `ept`. Предполагается, что число в строке записано с основанием, заданным в `base`
`unsigned long long strtoll( const char * restrict npt, char ** restrict ept, int base);`	То же, что и `strtol()`, но строка, указанная `npt`, преобразуется в тип `unsigned long long` (C99)
`int rand(void);`	Возвращает псевдослучайное число в диапазоне от `0` до `RAND_MAX`
`void srand(unsigned int seed);`	Устанавливает начальное значение для генератора случайных чисел в `seed`; если `rand()` вызывается до вызова `srand()`, `seed` равно `1`
`void *aligned_alloc( size_t algn, size_t size);`	Выделяет память для объекта выравнивания `algn` с размером `size` байтов; в `algn` должно быть указано поддерживаемое значение выравнивания, а значение `size` должно быть кратным `algn` (C11)

Продолжение табл. Б.V.36

Прототип	Описание
`void *calloc(size_t nmem, size_t size);`	Выделяет память для массива из nmem элементов, каждый элемент которого имеет размер size байтов; все биты в выделенной области инициализируются нулями. Функция возвращает адрес массива, если выполнена успешно, и NULL в противном случае
`void free(void *ptr);`	Освобождает память, на которую указывает ptr; указатель ptr должен иметь значение, ранее возвращенное вызовом calloc(), malloc() или realloc(), или же должен быть нулевым указателем — в этом случае никакие действия не предпринимаются. Для других значений указателя поведение не определено
`void *malloc(size_t size);`	Выделяет неинициализированный блок памяти размером в size байтов; при успешном завершении функция возвращает адрес блока памяти и NULL в противном случае
`void *realloc(void *ptr, size_t size);`	Изменяет размер блока памяти, на который указывает ptr, до size байтов; содержимое блока в части, меньшей из двух размеров, старого и нового, остается неизменным; функция возвращает местоположение блока, который может быть перемещен; если же память не может быть перераспределена, функция возвращает NULL и оставляет исходный блок незатронутым. Если ptr равно NULL, поведение аналогично вызову malloc() с аргументом size; если size равно нулю, а ptr не равен NULL, то поведение аналогично вызову free() с ptr в качестве аргумента
`void abort(void);`	Приводит к аварийному прекращению работы программы, если только не поступил сигнал SIGABRT и не возвращен соответствующий обработчик сигнала; закрытие потоков ввода-вывода и временных файлов зависит от реализации; функция выполняет raise(SIGABRT)
`int atexit( void (*func)(void));`	Регистрирует функцию, на которую указывает func, для вызова при нормальном завершении программы; реализация должна поддерживать регистрацию как минимум 32 функций, которые должны вызываться в порядке, противоположном порядку их регистрации; функция возвращает ноль, если регистрация удалась, и ненулевое значение — в противном случае
`int at_quick_exit( void (*func)(void));`	Регистрирует функцию, на которую указывает func, для вызова в случае вызова quick_exit(); реализация должна поддерживать регистрацию как минимум 32 функций, которые должны вызываться в порядке, противоположном порядку их регистрации; функция возвращает ноль, если регистрация удалась, и ненулевое значение — в противном случае (C11)
`void exit(int status);`	Приводит к нормальному завершению программы, сначала вызывая функции, которые зарегистрированы посредством atexit(). Затем она сбрасывает все открытые выходные потоки, закрывает все потоки ввода-вывода, закрывает все файлы, созданные tmpfile() и передает управление размещаемой среде. Если status равен 0 или EXIT_SUCCESS, то в размещаемую среду передается зависящий от реализации код возврата, обозначающий нормальное завершение. Если status равен EXIT_FAILURE, в размещаемую среду передается зависящий от реализации код, который обозначает неудачное завершение. Результат других значений status также зависит от реализации

Продолжение табл. Б.V.36

Прототип	Описание
`void _Exit(int status);`	Подобна `exit()`, но с тем отличием, что зарегистрированные посредством `atexit()` и `signal()` функции не вызываются, а обработка открытых потоков зависит от реализации (C99)
`char *getenv( const char * name);`	Возвращает указатель на строку, представляющую значение переменной среды, имя которой указано в `name`. Если переменной с таким именем нет, возвращает `NULL`
`_Noreturn void quick_exit( int status);`	Приводит к нормальному завершению программы. Функции, зарегистрированные с помощью `atexit()`, и обработчики сигналов, зарегистрированные посредством `signal()`, не вызываются. Функции, зарегистрированные с помощью `at_quick_exit()`, вызываются в порядке, обратном их регистрации. Поведение не будет определено, если программа вызывает `quick_exit()` более одного раза или вызывает обе функции `quick_exit()` и `exit()`. Управление возвращается размещаемой среде через вызов `_Exit(status)` (C11)
`int system(const char *str);`	Передает строку, указанную `str`, размещаемой среде для выполнения командным процессором, таким как DOS или UNIX. Если `str` равно `NULL`, функция возвращает ненулевое значение, когда командный процессор доступен, и нулевое — когда нет. Если значение `str` отлично от `NULL`, то возвращаемое значение зависит от конкретной реализации
`void *bsearch(const void *key, const void *base, size_t nmem, size_t size, int (*comp)(const void *, const void *));`	Выполняет поиск элемента, соответствующего объекту `key`, в массиве с `nmem` элементами размером `size`, на который указывает `base`. Элементы сравниваются с помощью функции `comp`. Функция сравнения должна возвращать значение меньше нуля, если ключевой объект меньше элемента массива, ноль, если они эквивалентны, и значение больше нуля, если ключевой объект больше. Функция возвращает указатель на найденный элемент или `NULL`, если элемент не найден. Если найдено два или более искомых элемента, то выбираемый из них элемент не регламентируется
`void qsort(void *base, size_t nmem, size_t size, int (*comp) (const void *, const void *));`	Сортирует массив, указанный `base`, в порядке, устанавливаемом функцией `comp`; массив состоит из `nmem` элементов, каждый из которых имеет размер `size` байтов. Функция сравнения должна возвращать значение меньше нуля, если объект, указанный в ее первом аргументе, меньше, чем объект, указанный вторым аргументом, ноль, если объекты эквивалентны, и значение больше нуля, если первый объект больше
`int abs(int n);`	Возвращает абсолютное значение `n`; возвращаемое значение может быть неопределенным, если `n` — отрицательная величина, не имеющая положительного аналога, что может произойти, когда `n` равно `INT_MIN` в представлении с дополнением до двух
`div_t div(int numer, int denom);`	Вычисляет частное и остаток от деления `numer` на `denum`, помещая частное в элемент `quot` структуры `div_t`, а остаток от деления — в элемент `rem` этой структуры. При неточном делении частное равно целому значению наименьшей величины, ближайшей к алгебраическому частному (т.е. производится округление в сторону нуля)

Продолжение табл. Б.V.36

Прототип	Описание
`long labs(int n);`	Возвращает абсолютное значение n; возвращаемое значение может быть неопределенным, если n — отрицательная величина, не имеющая положительного аналога, что может произойти, когда n равно LONG_MIN в представлении с дополнением до двух
`ldiv_t ldiv(long numer, long denom);`	Вычисляет частное и остаток от деления numer на denum, помещая частное в элемент quot структуры div_t, а остаток от деления — в элемент rem этой структуры. При неточном делении частное равно целому значению наименьшей величины, ближайшей к алгебраическому частному (т.е. производится округление в сторону нуля).
`long long llabs(int n);`	Возвращает абсолютное значение n; возвращаемое значение может быть неопределенным, если n — отрицательная величина, не имеющая положительного аналога, что может произойти, когда n равно LONG_LONG_MIN в представлении с дополнением до двух (C99)
`lldiv_t lldiv(long numer, long denom);`	Вычисляет частное и остаток от деления numer на denum, помещая частное в элемент quot структуры div_t, а остаток от деления — в элемент rem этой структуры. При неточном делении частное равно целому значению наименьшей величины, ближайшей к алгебраическому частному (т.е. производится округление в сторону нуля) (C99)
`int mblen(const char *s, size_t n);`	Возвращает количество байтов (до n), из которых состоит многобайтный символ, указанный s. Возвращает 0, если s указывает на нулевой символ, и -1, если s не указывает на многобайтный символ. Если s равно NULL, возвращает ненулевое значение, когда многобайтные символы имеют зависящую от состояния кодировку, или ноль — в противном случае
`int mbtowc(wchar_t *pw, const char *s, size_t n);`	Если s — не NULL, определяет количество байтов (до n), составляющих многобайтный символ, на который указывает s, и определяет код символа типа wchar_t. Если pw не равно NULL, записывает код в место, указанное pw; возвращает то же значение, что и mblen(s, n)
`int wctomb(char *s, wchar_t wc);`	Преобразует код символа wc в соответствующее представление многобайтного символа и сохраняет его в массиве, на который указывает s, если только s не равно NULL. Если s не равно NULL, возвращает -1, когда wc не соответствует корректному многобайтному символу. Если же wc корректно, возвращает количество байтов, составляющих многобайтный символ. Если s равно NULL, возвращает ненулевое значение, когда многобайтный символ имеет зависящую от состояния кодировку, и ноль — в противном случае
`size_t mbstowcs( wchar_t * restrict pwcs, const char *s restrict, size_t n);`	Преобразует массив многобайтных символов, указанный s, в массив кодов широких символов и сохраняет его в месте, указанном pwcs. Преобразование выполняется максимум для n элементов массива pwcs либо до появления нулевого байта в массиве s, в зависимости от того, что произойдет раньше. Если встречается некорректный многобайтный символ, возвращает (size_t) (-1); в противном случае возвращает количество заполненных элементов массива (за исключением нулевого символа, если он есть)

Окончание табл. Б.V.36

Прототип	Описание
`size_t wcstombs( char * restrict s, const wchart_t * restrict pwcs, size_t n);`	Преобразует последовательность кодов широких символов в массиве, указанном pwcs, в последовательность многобайтных символов и копирует ее в место, указанное s. Обработка прекращается либо до сохранения n байт, либо до появления нулевого символа — в зависимости от того, что произойдет раньше. Если встречается некорректный многобайтный символ, возвращает (size_t) (-1); в противном случае возвращает количество заполненных элементов массива (за исключением нулевого символа, если он есть)

_Noreturn: stdnoreturn.h

Это определяет макрос noreturn, который расширяется до _Noreturn.

Обработка строк: string.h

В заголовочном файле string.h определен тип size_t и макрос NULL для нулевого указателя. Кроме того, предлагается ряд функций для анализа и манипулирования символьными строками, а также несколько функций, работающих с памятью более универсальным способом. Эти функции перечислены в табл. Б.V.37.

Таблица Б.V.37. Строковые функции

Прототип	Описание
`void *memchr(const void *s, int c, size_t n);`	Ищет первое вхождение c (преобразованное в unsigned char) в первых n символах объекта, на который указывает s; возвращает указатель на первое вхождение или NULL, если ничего не найдено
`int memcmp(const void *s1, const void *s2, size_t n);`	Сравнивает первые n символов объекта, на который указывает s1, с первыми n символами объекта, на который указывает s2, интерпретируя каждый символ как unsigned char. Два объекта идентичны, если все n пар совпадают; в противном случае объекты сравниваются по первой несовпадающей паре. Возвращает ноль, если объекты идентичны, значение меньше нуля, если первый объект в числовом виде меньше второго, и значение больше нуля, если первый объект больше второго
`void *memcpy(void * restrict s1, const void * restrict s2, size_t n);`	Копирует n байтов из места, указанного s2, в место, указанное s1; поведение не определено, если две области перекрываются; возвращает значение s1
`void *memmove(void *s1, const void *s2, size_t n);`	Копирует n байтов из места, указанного s2, в место, указанное s1; поведение аналогично memmove(). Сначала использует временное место, поэтому перекрывающееся копирование допускается; возвращает значение s1
`void *memset(void *s, int v, size_t n);`	Копирует значение v (преобразованное в тип unsigned char) в первые n байтов, которые находятся по адресу, указанному s; возвращает s
`char *strcat(char * restrict s1, const char * restrict s2);`	Добавляет копию строки, на которую указывает s2 (включая нулевой символ), в место, указанное s1; при этом первый символ s2 перекрывает нулевой символ s1; возвращает s1

Продолжение табл. Б.V.37

Прототип	Описание
`char *strncat(char * restrict s1, const char * restrict s2, size_t n);`	Добавляет копию до n символов или до появления нулевого символа строки, указанной s2, в место, указанное s1; при этом первый символ s2 перекрывает нулевой символ s1; всегда добавляется нулевой символ; функция возвращает s1
`char *strcpy(char * restrict s1, const char * restrict s2);`	Копирует строку, указанную s2 (включая нулевой символ), в место, указанное s1; возвращает s1
`char *strncpy(char * restrict s1, const char * restrict s2, size_t n);`	Копирует до n символов или до появления нулевого символа из строки, указанной s2, в место, указанное s1. Если в s2 встретится нулевой символ до того, как будут скопированы n символов, то функция дописывает нулевые символы, дополняющие до общего количества n. Если же n символов будут скопированы до того, как встретится нулевой символ, то никакого нулевого символа не добавляется. Строка возвращает s1
`int strcmp(const char *s1, const char *s2);`	Сравнивает строки, указанные s1 и s2. Две строки идентичны, если совпадают все пары соответствующих символов; иначе строки сравниваются по первой несовпадающей паре. Символы сравниваются по значениям их кодов; функция возвращает ноль, если строки идентичны, отрицательное значение, если первая строка меньше второй, и положительное — если первая строка больше второй
`int strcoll(const char *s1, const char *s2);`	Работает подобно strcmp(), но использует последовательность сопоставления, описанную категорией LC_COLLATE текущей локальной установки, которая задана функцией setlocale()
`int strncmp(const char *s1, const char *s2, size_t n);`	Сравнивает до n первых символов или до появления нулевого символа в массивах, указанных s1 и s2; два массива идентичны, если все проверенные пары символов совпадают; иначе массивы сравниваются по первой несовпадающей паре. Символы сравниваются по значениям их кодов; функция возвращает ноль, если массивы идентичны, значение меньше нуля, если первый массив меньше второго, и значение больше нуля, если первый массив больше второго
`size_t strxfrm( char * restrict s1, const char * restrict s2, size_t n);`	Трансформирует строку s2 и копирует до n символов, включая завершающий нулевой символ, в массив, указанный s1. Критерием трансформации является то, что трансформированные строки должны упорядочиваться функцией strcmp() в том же порядке, в каком strcol() размещала бы их нетрансформированные версии. Функция возвращает длину трансформированной строки (исключая нулевой символ)
`char *strchr(const char *s, int c);`	Ищет первое вхождение символа c (преобразованного в char) в строке, на которую указывает s; нулевой символ рассматривается как часть строки; возвращает указатель на первое вхождение или NULL, если символ не найден
`size_t strcspn(const char *s1, const char *s2);`	Возвращает длину максимального начального сегмента s1, который не содержит ни одного символа из s2
`char *strpbrk(const char *s1, const char *s2);`	Возвращает указатель на первый символ s1, совпадающий с любым из символов s2; возвращает NULL, если соответствие не обнаружено

Окончание табл. Б.V.37

Прототип	Описание
`char *strrchr(const char *s, int c);`	Ищет последнее вхождение символа `c` (преобразованного в `char`) в строке, на которую указывает `s`; нулевой символ рассматривается как часть строки; возвращает указатель на первое вхождение или `NULL`, если символ не найден
`size_t strspn(const char *s1, const char *s2);`	Возвращает длину максимального начального сегмента `s1`, который полностью состоит из символов, входящих в `s2`
`char *strstr(const char *s1, const char *s2);`	Возвращает указатель на место первого вхождения в `s1` последовательности символов `s2` (исключая нулевой символ); возвращает `NULL`, если вхождение не обнаружено
`char *strtok(char * restrict s1, const char * restrict s2);`	Эта функция разбивает строку `s1` на отдельные лексемы. Строка `s2` содержит символы, служащие разделителями лексем. Функция вызывается последовательно. При начальном вызове `s1` должно указывать на строку, которую требуется разбить на лексемы. Функция находит первый разделитель, который следует за символом, не являющимся разделителем, и заменяет его нулевым символом. Она возвращает указатель на строку, содержащую первую лексему. Если никаких лексем не найдено, возвращается `NULL`. Чтобы найти последующие лексемы строки, `strtok()` нужно вызвать снова, но в качестве первого аргумента указать `NULL`. Каждый последующий вызов возвращает указатель на следующую лексему или `NULL`, если больше лексем не найдено. (Сразу после таблицы приводится пример.)
`char * strerror(int errnum);`	Возвращает указатель на зависящую от реализации строку сообщения об ошибке, которая соответствует номеру ошибки, переданному в `errnum`
`int strlen(const char * s);`	Возвращает количество символов (исключая нулевой) в строке `s`

Функция `strtok()` применяется несколько необычно, поэтому рассмотрим небольшой пример:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char data[] = "  C приносит\t  очень#много\nрадости!";
   const char tokseps[] = " \t\n#";     /* разделители        */
   char * pt;

   puts(data);
   pt = strtok(data,tokseps);            /* первый вызов       */
   while (pt)                            /* выход в случае NULL */
   {
      puts (pt);                         /* показать лексему   */
      pt = strtok(NULL, tokseps);        /* следующая лексема  */
   }
   return 0;
}
```

Вот вывод, полученный в результате запуска этого примера:

```
  С приносит   очень#много
радости!
С
приносит
очень
много
радости!
```

Математические функции для обобщенных типов: tgmath.h (C99)

Библиотеки math.h и complex.h предоставляют множество экземпляров функций, которые отличаются только типами. Например, все шесть функций, показанные ниже, вычисляют синус:

```
double sin(double);
float sinf(float);
long double sinl(long double);
double complex csin(double complex);
float csinf(float complex);
long double csinl(long double complex);
```

В заголовочном файле tgmath.h определены макросы, которые расширяют обобщенный вызов в обращение к соответствующей функции, как указано типом аргумента. Следующий код иллюстрирует использование макроса sin(), расширяемого в различные формы функции синуса:

```
#include <tgmath.h>
...
double dx, dy;
float fx, fy;
long double complex clx, cly;
dy = sin(dx);         // расширяется в dy = sin(dx)  (функция)
fy = sin(fx);         // расширяется в fy = sinf(fx)
cly = sin(clx);       // расширяется в cly = csinl(clyx)
```

В этом заголовочном файле определены обобщенные макросы для трех классов функций. Первый класс состоит из функций math.h и complex.h, определенных в шести вариациях, в которых применяются суффиксы f и l и префикс c, как в предыдущем примере с sin(). В данном случае обобщенный макрос носит то же имя, что и версия double функции.

Во второй класс входят функции math.h, определенные в трех вариациях с использованием суффиксов f и l, которые не имеют комплексных аналогов, к примеру, erf(). В этом случае имя макроса выглядит так же, как имя функции без суффикса, в рассматриваемом примере — erf(). Результат применения такого макроса с комплексным аргументом не определен.

Третий класс состоит из функций complex.h, которые определены в трех вариациях с использованием суффиксов f и l и не имеют вещественных аналогов, таких как cimag(). В данном случае имя макроса совпадает с именем функции без суффикса, в этом примере — cimag(). Результат применения такого макроса с вещественным аргументом не определен.

В табл. Б.V.38 перечислены обобщенные функциональные макросы.

Таблица Б.V.38. Обобщенные математические функции

acos	asin	atanb	acosh	asinh	atanh
cos	sin	tan	cosh	sinh	tanh
exp	log	pow	sqrt	fabs	atan2
cbrt	ceil	copysign	erf	erfc	exp2
expml	fdim	floor	fma	fmax	fmin
fmod	frexp	hypot	ilogb	ldexp	lgamma
llrint	llround	log10	log1p	log2	logb
lrint	lround	nearbyint	nextafter	nexttoward	remainder
remquo	rint	round	scalbn	scalbln	tgamma
trunc	carg	cimag	conj	cproj	creal

До выхода стандарта C11 для определения обобщенных макросов реализации должны были прибегать к расширениям стандарта. Но добавление выражения _Generic делает возможной прямую реализацию с использованием стандарта C11.

Потоки: threads.h (C11)

Заголовочный файл threads.h наряду с stdatomic.h предоставляет поддержку для параллельного программирования. Эта тема выходит за рамки настоящей книги, но, выражаясь общими терминами, данный заголовочный файл поддерживает множество потоков выполнения, которые в принципе могут быть назначены разным процессорам.

Дата и время: time.h

В заголовочном файле time.h определены три макроса. Первым из них, который также определен во многих других заголовочных файлах, является NULL, представляющий нулевой указатель. Второй макрос — это CLOCKS_PER_SEC; деление на этот макрос значения, возвращенного функцией clock(), позволяет получить время в секундах. Третий макрос (C11) называется TIME_UTC и представляет собой положительную целочисленную константу, обозначающую координату времени UTC (Universal Time Coordinated — универсальное синхронизированное время), которая является потенциальным аргументом функции timespec_get().

UTC — это текущий основной стандарт мирового времени. Он применяется, например, в авиации, при составлении прогнозов погоды, для синхронизации компьютерных часов и в качестве общего стандарта в Интернете.

Определенные в этом заголовочном файле типы перечислены в табл. Б.V.39.

Таблица Б.V.39. Типы, определенные в time.h

Тип	Описание
size_t	Целочисленный тип, возвращаемый операцией sizeof
clock_t	Арифметический тип, подходящий для представления времени
time_t	Арифметический тип, подходящий для представления времени
struct timespec	Тип структуры для хранения интервала времени, указанного в секундах и наносекундах (C11)
struct tm	Тип структуры для хранения компонентов календарного времени

Структура timespec содержит, по меньшей мере, два члена, показанные в табл. Б.V.40.

Таблица Б.V.40. Члены структуры timespec

Член	Описание
`time_t tv_sec`	Полное число секунд (>=0)
`long tv_nsec`	Количество наносекунд (из диапазона [0, 999999999])

Компоненты календарного типа называют *разделенным на составляющие временем.* В табл. Б.V.41 перечислены обязательные члены структуры `struct tm`.

Таблица Б.V.41. Члены структуры struct tm

Член	Описание
`int tm_sec`	Секунды после минут (0–61)
`int tm_min`	Минуты после часов (0–59)
`int tm_hour`	Часы после полуночи (0–23)
`int tm_mday`	Дни месяца (1–31)
`int tm_mon`	Месяцы, начиная с января (0–11)
`int tm_year`	Годы, начиная с 1900
`int tm_wday`	Дни, начиная с воскресенья (0–6)
`int tm_yday`	Дни, начиная с 1 января (0–365)
`int tm_isdst`	Флаг перехода на летнее время (больше нуля — значит, переход на летнее время включен; ноль — нет; отрицательное значение говорит о том, что информация недоступна)

Термин *календарное время* обозначает текущую дату и время; например, это может быть количество секунд, прошедших после первой секунды 1900 года. Термин *локальное время* – это календарное время, выраженное для локального часового пояса. Функции для работы со временем перечислены в табл. Б.V.42.

Таблица Б.V.42. Функции для работы со временем

Прототип	Описание
`clock_t clock(void);`	Возвращает наилучшее для данной реализации приближение процессорного времени, прошедшего с момента запуска программы; для получения времени в секундах значение необходимо разделить на `CLOCKS_PER_SEC`. Если время не доступно или не может быть представлено, возвращает `(clock_t)(-1)`
`double difftime(time_t t1, time_t t0);`	Вычисляет разницу `(t1 - t0)` между двумя значениями календарного времени; выражает результат в секундах и возвращает его
`time_t mktime( struct tm *tmptr);`	Преобразует разделенное на составляющие время, представленное в структуре, на которую указывает `tmptr`, в календарное время. Используя ту же кодировку, что и функция `time()`, структура изменяется таким образом, что значения, выходящие за диапазон допустимых, корректируются (например, 2 минуты и 100 секунд становятся 3 минутами и 40 секундами), а `tm_wday` и `tm_yday` устанавливаются в величины, которые вытекают из значений остальных элементов структуры. Если календарное время не может быть правильно представлено, возвращает `(time_t)(-1)`, иначе возвращает календарное время в формате `time_t`

Окончание табл. Б.V.42

Прототип	Описание
`time_t time(time_t *ptm)`	Возвращает текущее календарное время и помещает его в место, на которое указывает `ptm`, предполагая, что `ptm` не равно `NULL`. Если календарное время не доступно, возвращает `(time_t)(-1)`
`int timespec_get( struct timespec * ts, int base)`	Устанавливает структуру, на которую указывает `ts`, в текущее календарное время на основе указанных координат времени. Возвращает `base` (ненулевое значение) при успешной установке и ноль в противном случае (C11)
`char *asctime( const struct tm *tmpt);`	Преобразует разделенное на составляющие время из структуры, указанной с помощью `tmpt`, в строку вида `Sun Feb 15 13:14:33 2015\n\0` и возвращает указатель на эту строку
`char *ctime( const time_t *ptm);`	Преобразует календарное время из структуры, указанной `ptm`, в строку вида `Wed Feb 11 10:54:47 2015\n\0` и возвращает указатель на эту строку
`struct tm *gmtime( const time_t *ptm);`	Преобразует календарное время из структуры, указанной `ptm`, в разделенное на составляющие время, выраженное как универсальное глобальное время (UTC), предшественник времени по Гринвичу (GMT), и возвращает указатель на структуру с этой информацией. Если UTC недоступно, возвращает `NULL`
`struct tm *localtime( const time_t *ptm);`	Преобразует календарное время, указанное `ptm`, в разделенное на составляющие время, выраженное как местное. Сохраняет его в структуре `tm` и возвращает указатель на нее
`size_t strftime( char * restrict s, size_t max const char * restrict fmt, const struct tm * restrict tmpt);`	Копирует строку `fmt` в строку `s` с заменой спецификаторов формата (табл. Б.V.43) в `fmt` соответствующими данными, взятыми из структуры разделенного на составляющие времени, на которую указывает `tmpt`; в строку `s` помещается не более `max` символов. Функция возвращает количество символов (исключая нулевой) в результирующей строке. Если результирующая строка (включая нулевой символ) содержит больше, чем `max` символов, функция возвращает `0`, а содержимое `s` не определено

В табл. Б.V.43 представлены спецификаторы формата, применяемые в функции `strftime()`. Многие заменяемые значения, такие как названия месяцев, зависят от текущей локальной установки.

Таблица Б.V.43. Спецификаторы формата, используемые в функции `strftime()`

Спецификатор формата	Чем заменяется
`%a`	Локальное сокращенное название дня недели
`%A`	Локальное полное название дня недели
`%b`	Локальное сокращенное название месяца
`%B`	Локальное полное название месяца
`%c`	Локальный разделитель даты и времени
`%d`	День месяца в виде десятичного числа (01-31)
`%D`	Эквивалент `"%m/%d%y"`
`%e`	День месяца в виде десятичного числа — однозначные числа предваряются пробелом

Окончание табл. Б.V.43

Спецификатор формата	Чем заменяется
%F	Эквивалент "%Y-%m-%d"
%g	Последние две цифры года (00-99)
%G	Год в виде десятичного числа
%h	Эквивалент "%b"
%H	Часы (по 24-часовой шкале) в виде десятичного числа (00-23)
%I	Часы (по 12-часовой шкале) в виде десятичного числа (01-12)
%j	День года в виде десятичного числа (001-366)
%m	Месяц в виде десятичного числа (01-12)
%n	Символ новой строки
%M	Минуты в виде десятичного числа (00-59)
%p	Локальный эквивалент "до полудня/после полудня" (a.m./p.m.) для 12-часовой временной шкалы
%r	Локальное 12-часовое время
%R	Эквивалент "%H:%M"
%S	Секунды в виде десятичного числа (00-61)
%t	Символ горизонтальной табуляции
%T	Эквивалент "%H:%M:%S"
%u	Номер дня недели по ISO 8601 (1-7), где 1 соответствует понедельнику
%U	Номер недели в году, считая воскресенье первым днем недели 1 (00-53)
%V	Номер недели в году в соответствие с ISO 8601, считая воскресенье первым днем недели 1 (00-53)
%w	Номер дня недели в виде десятичного числа, начиная с воскресенья (0-6)
%W	Номер недели в году, считая понедельник первым днем недели 1 (00-53)
%x	Локальное представление даты
%X	Локальное представление времени
%y	Год без века в виде десятичного числа (00-99)
%Y	Год с веком в виде десятичного числа
%z	Смещение от UTC в формате ISO 8601 ("-800" означает восемь часов от Гринвичу, т.е. на восемь часов западнее); если информация недоступна, то никакие символы не подставляются
%Z	Название часового пояса; если информация недоступна, то никакие символы не подставляются
%%	% (т.е. знак процента)

Утилиты Unicode: uchar.h (C11)

Заголовочный файл wchar.h из C99 предлагает два средства поддержки крупных наборов символов. В C11 добавлена поддержка, специально ориентированная на Unicode, за счет предоставления типов, который подходят для кодировки UTF-16 и UTF-32 (табл. Б.V.44).

Таблица Б.V.44. Типы, объявленные в uchar.h

Тип	Описание
char16_t	Целочисленный тип без знака, используемый для 16-битовых символов (тот же самый тип, что и uint_least16_t из stdint.h)
char32_t	Целочисленный тип без знака, используемый для 32-битовых символов (тот же самый тип, что и uint_least32_t из stdint.h)
size_t	Целочисленный тип, возвращаемый операцией sizeof (stddef.h)
mbstate_t	Тип, отличный от массива, который может хранить информацию о состоянии преобразования, необходимую для выполнения преобразований между последовательностями многобайтных символов и широких символов (wchar.h)

В этом заголовочном файле объявлены функции (табл. Б.V.45) для преобразования строк многобайтных символов в форматы chart16_t и char32_t и наоборот.

Таблица Б.V.45. Функции для выполнения преобразований между широкими символами и многобайтными символами

Прототип	Описание
size_t mbrto16(char16_t * restrict pwc, const char * restrict s, size_t n, mbstate_t * restrict ps);	То же, что и mbrtowc() (wchar.h), но преобразует в тип char_16, а не wchar_t
size_t mbrto32(char32_t * restrict pwc, const char * restrict s, size_t n, mbstate_t * restrict ps);	То же, что и mbrto16(), но преобразует в тип char32_t
size_t c16rtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);	То же, что и wcrtobm() (wchar.h), но преобразует из типа char16_t, а не wchar_t
size_t c32rtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);	То же, что и wcrtobm() (wchar.h), но преобразует из типа char32_t, а не wchar_t

Утилиты для работы с многобайтными и широкими символами: wchar.h (C99)

Каждая реализация имеет свой базовый набор символов, и тип char в языке C должен быть достаточно широким, чтобы поддерживать этот набор. Реализация может также поддерживать расширенные наборы символов, а эти символы могут требовать для своего представления более одного байта на символ. Многобайтные символы могут сохраняться наряду с однобайтными в обычном массиве char, где определенные значения байта служат признаками присутствия многобайтного символа и его размера. Интерпретация многобайтных символов может зависеть от *состояния сдвига*. В начальном состоянии сдвига однобайтные символы интерпретируются обычным образом. Специфические многобайтные символы затем могут изменять состояние сдвига. Определенное состояние сдвига остается в силе до тех пор, пока не будет явно изменено.

Тип wchar_t обеспечивает второй способ представления широких символов, когда ширина типа выбирается достаточной для представления кодировки любого элемента из расширенного набора символов. Такое представление широких символов позволяет помещать отдельные символы в переменные типа wchar_t, а строки таких символов представлять в виде массивов wchar_t. Представление широких символов не обязательно должно совпадать с многобайтным представлением, потому что второе может применять состояния сдвига, в то время как первое — нет.

Заголовочный файл wchar_t предоставляет средства для обработки обоих представлений широких символов. В нем определены типы, перечисленные в табл. Б.V.46 (некоторые из этих типов также определены в других заголовочных файлах).

Таблица Б.V.46. Типы, определенные в wchar.h

Тип	Описание
wchar_t	Целочисленный тип, который может представить наибольший набор символов, указанный поддерживаемыми локальными установками
wint_t	Целочисленный тип, который может вместить любое значение из расширенного набора символов плюс, по крайней мере, одно значение, не входящее в расширенный набор символов
size_t	Целочисленный тип, возвращаемый операцией sizeof
mbstate_t	Тип, отличный от массива, который может хранить информацию о состоянии преобразования, необходимую для выполнения преобразований между последовательностями многобайтных символов и широких символов
struct tm	Тип структуры для хранения компонентов календарного времени

В заголовочном файле также определено несколько макросов, которые приведены в табл. Б.V.47.

Таблица Б.V.47. Макросы, определенные в wchar.h

Макрос	Описание
NULL	Нулевой указатель
WCHAR_MAX	Максимальное значение wchar_t
WCHAR_MIN	Минимальное значение wchar_t
WEOF	Константное выражение типа wchar_t, которое не соответствует ни одному элементу в расширенном наборе символов; это эквивалент EOF в широких символах, используемый для индикации состояния конца файла при вводе широких символов

Библиотека содержит функции ввода-вывода, являющиеся аналогами стандартных функций ввода-вывода, которые определены в stdio.h. В тех случаях, когда стандартная функция ввода-вывода возвращает EOF, соответствующая функция для широких символов возвращает WEOF. Эти функции перечислены в табл. Б.V.48.

Таблица Б.V.48. Функции ввода-вывода для работы с широкими символами

Прототип функции
`int fwprintf(FILE * restrict stream, const wchar_t * restrict format, ...);`
`int fwscanf(FILE * restrict stream,const wchar_t * restrict format, ...);`
`int swprintf(wchar_t * restrict s, size_t n,const wchar_t * restrict format, ...);`
`int swscanf(const wchar_t * restrict s,const wchar_t * restrict format, ...);`
`int vfwprintf(FILE * restrict stream, const wchar_t * restrict format, va_list arg);`
`int vfwscanf(FILE * restrict stream, const wchar_t * restrict format, va_list arg);`
`int vswprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, va_list arg);`
`int vswscanf(const wchar_t * restrict s, const wchar_t * restrict format, va_list arg);`
`int vwprintf(const wchar_t * restrict format, va_list arg);`
`int vwscanf(const wchar_t * restrict format, va_list arg);`
`int wprintf(const wchar_t * restrict format, ...);`
`int wscanf(const wchar_t * restrict format, ...);`
`wint_t fgetwc(FILE *stream);`
`wchar_t *fgetws(wchar_t * restrict s, int n, FILE * restrict stream);`
`wint_t fputwc(wchar_t c, FILE *stream);`
`int fputws(const wchar_t * restrict s, FILE * restrict stream);`
`int fwide(FILE *stream, int mode);`
`wint_t getwc(FILE *stream);`
`wint_t getwchar(void);`
`wint_t putwc(wchar_t c, FILE *stream);`
`wint_t putwchar(wchar_t c);`
`wint_t ungetwc(wint_t c, FILE *stream);`

Существует одна функция ввода-вывода с широкими символами, не имеющая аналога в стандартной библиотеке ввода-вывода:

```
int fwide(FILE *stream, int mode);
```

Если аргумент `mode` имеет положительное значение, то сначала эта функция пытается трактовать поток, представленный параметром `stream`, как *ориентированный на широкие символы*, а если отрицательное — то как *ориентированный на байты*. Если же значение `mode` равно нулю, функция не пытается изменить ориентацию потока. Попытка изменения ориентации предпринимается, только если она не была назначена потоку изначально. Во всех случаях функция возвращает положительное значение, если поток ориентирован на широкие символы, отрицательное значение, если поток ориентирован на байты, и ноль, если ориентация потока не установлена.

Заголовочный файл `wchar.h` предлагает несколько функций для манипуляции и преобразования строк, которые моделируют такие же функции из `string.h`. В общем случае фрагмент `str` в идентификаторах из `string.h` заменяется фрагментом `wcs`, так что `wcstod()` — это версия функции `strtod()` для широких символов. Такие функции перечислены в табл. Б.V.49.

Таблица Б.V.49. Строковые утилиты для широких символов

Прототип функции
`double wcstod(const wchar_t * restrict nptr, wchar_t ** restrict endptr);`
`float wcstof(const wchar_t * restrict nptr, wchar_t ** restrict endptr);`
`long double wcstold(const wchar_t * restrict nptr, wchar_t ** restrict endptr);`
`long int wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);`
`long long int wcstoll(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);`
`unsigned long int wcstoul(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);`
`unsigned long long int wcstoull(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);`
`wchar_t *wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2);`
`wchar_t *wcsncpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);`
`wchar_t *wcscat(wchar_t * restrict s1, const wchar_t * restrict s2);`
`wchar_t *wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);`
`int wcscmp(const wchar_t *s1, const wchar_t *s2);`
`int wcscoll(const wchar_t *s1, const wchar_t *s2);`
`int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);`
`size_t wcsxfrm(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);`
`wchar_t *wcschr(const wchar_t *s, wchar_t c);`
`size_t wcscspn(const wchar_t *s1, const wchar_t *s2);`
`size_t wcslen(const wchar_t *s);`
`wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);`
`wchar_t *wcsrchr(const wchar_t *s, wchar_t c);`
`size_t wcsspn(const wchar_t *s1, const wchar_t *s2);`
`wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);`
`wchar_t *wcstok(wchar_t * restrict s1, const wchar_t * restrict s2, wchar_t ** restrict ptr);`
`wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);`
`int wmemcmp(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);`
`wchar_t *wmemcpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);`
`wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);`
`wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);`

В этом заголовочном файле также объявлена функция, моделирующая `strftime()` из `time.h`:

```
size_t wcsftime(wchar_t * restrict s, size_t maxsize,
const wchar_t * restrict format,
const struct tm * restrict timeptr);
```

И, наконец, здесь объявлено несколько функций для преобразования строк с широкими символами в строки с многобайтными символами и наоборот (табл. Б.V.50).

Таблица Б.V.50. Функции для выполнения преобразований между широкими символами и многобайтными символами

Прототип	Описание
`wint_t btowc(int c);`	Если (unsigned char) c является допустимым однобайтным символом в начальном состоянии сдвига, то функция возвращает его представление в широких символах, иначе возвращает WEOF
`int wctob(wint_t c);`	Если c является элементом расширенного набора символов, многобайтное символьное представление которого в начальном состоянии сдвига представлено одним байтом, то функция возвращает однобайтное представление unsigned char, преобразованное в int; иначе функция возвращает EOF
`int mbsinit(const mbstate_t *ps);`	Функция возвращает ненулевое значение, если ps — нулевой указатель или указывает на объект данных, описывающий начальное состояние преобразования; в противном случае функция возвращает ноль
`size_t mbrlen( const char * restrict s, size_t n, mbstate_t * restrict ps);`	Функция mbrlen() эквивалентна вызову mbrtowc(NULL, s, n, ps != NULL ? ps : &internal), где internal — объект mbstate_t для функции mbrlen(), за исключением того, что выражение, назначенное ps, оценивается только один раз
`size_t mbrtowc( wchar_t * restrict pwc, const char * restrict s, size_t n, mbstate_t * restrict ps);`	Если s является нулевым указателем, то вызов эквивалентен установке pwc в нулевой указатель, а n — в 1. Если же s — ненулевой указатель, то функция инспектирует максимум n байтов с целью определения количества байтов, необходимых для завершения следующего многобайтного символа (включая любые последовательности сдвига). Если функция определяет, что следующий многобайтный символ завершен и корректен, она определяет значение соответствующего широкого символа и затем, если pwc — ненулевой указатель, сохраняет это значение в объекте, на который указывает pwc. Если соответствующий широкий символ является нулевым, то результирующее состояние описывается как начальное состояние преобразования. Функция возвращает 0, если обнаружен нулевой широкий символ. Если же обнаружен другой допустимый широкий символ, она возвращает количество байтов, необходимых для завершения символа. Если n байт недостаточно для того, чтобы описать допустимый широкий символ, и потенциально возможно его частичное представление, то функция возвращает -2. Если обнаружена ошибка кодирования, функция возвращает -1, записывает EILSEQ в errno и ничего не сохраняет
`size_t wcrtomb( char * restrict s, wchar_t wc, mbstate_t * restrict ps);`	Если s — нулевой указатель, то этот вызов эквивалентен установке wc в широкий нулевой символ и использованию внутреннего буфера для первого аргумента. Если s — ненулевой указатель, то функция wcrtomb() определяет количество байт, необходимое для представления многобайтного символа, который соответствует широкому символу, заданному wc (включая любые последовательности сдвига), и сохраняет многобайтное представление в массиве, на первый элемент которого указывает s. Сохраняется максимум MB_CUR_MAX символов. Если wc — широкий нулевой символ, то сохраняется нулевой байт, которому предшествует последовательность сдвига, необходимая для восстановления начального состояния сдвига; результирующее состояние описывает начальное состояние преобразования. Если wc — допустимый широкий символ, то функция возвращает количество байтов, необходимых для сохранения многобайтной версии, включая байты, описывающие состояние сдвига (если они есть). Если wc не является допустимым, функция записывает EILSEQ в errno и возвращает -1

Окончание табл. Б.V.50

Прототип	Описание
`size_t mbsrtowcs( wchar_t * restrict dst, const char ** restrict src, size_t len, mbstate_t * restrict ps);`	Функция `mbstrtowcs()` преобразует последовательность многобайтных символов, которая начинается в состоянии преобразования, описанном объектом, на который указывает `ps`, из массива, косвенно указанного `src`, в последовательность соответствующих широких символов. Если `dst` — ненулевой указатель, преобразованные символы сохраняются в массиве, на который указывает `dst`. Преобразование продолжается вплоть до завершающего нулевого символа, включая его. Преобразование прекращается раньше в двух случаях: когда встречается последовательность байтов, которая не может формировать допустимый многобайтный символ, и (если `dst` — ненулевой символ) когда `len` широких символов уже сохранено в массив, указанный с помощью `dst`. Каждое преобразование выполняется, как будто оно инициировано вызовом функции `mbrtowc()`. Если `dst` — ненулевой указатель, то объекту указателя, на который указывает `src`, присваивается нулевое значение (если преобразование остановилось по достижении нулевого символа) или адрес, следующий сразу за последним преобразованным многобайтным символом (при его наличии). Если же преобразование прекратилось по достижении нулевого символа и `dst` — ненулевой указатель, то результирующее состояние описывается как начальное состояние преобразования. В случае успеха функция возвращает количество преобразованных многобайтных символов (исключая нулевой, если он есть); в противном случае функция возвращает `-1`
`size_t wcsrtombs( char * restrict dst, const wchar_t ** restrict src, size_t len, mbstate_t * restrict ps);`	Функция `wcsrtombs()` преобразует последовательность широких символов из массива, косвенно указанного `src`, в последовательность соответствующих многобайтных символов, которые начинаются в состоянии преобразования, описанном объектом, на который указывает `ps`. Если `dst` — ненулевой указатель, то преобразованные символы затем сохраняются в массиве, указанном `dst`. Преобразование продолжается вплоть до нулевого символа, включая его, а прекращается раньше в двух случаях: когда встречается широкий символ, который не соответствует допустимому многобайтному символу, и (если `dst` — ненулевой символ) когда следующий многобайтный символ должен превысить лимит общего количества `len` байтов, которые должны быть сохранены в массиве, указанном `dst`. Каждое преобразование выполняется, как если бы была вызвана функция `wcrtomb()`. Если `dst` — ненулевой символ, то объекту указателя, на который указывает `src`, присваивается либо нулевое значение (если преобразование остановлено по достижении нулевого широкого символа), либо адрес символа, следующего сразу за последним преобразованным широким символом (если он был). Если же преобразование остановлено по достижении нулевого широкого символа, то результирующее состояние описывается как начальное состояние преобразования. В случае успеха функция возвращает количество многобайтных символов в результирующей многобайтной последовательности (исключая нулевой символ, если он есть); в противном случае возвращается `-1`

Утилиты классификации и отображения широких символов: wctype.h (C99)

Заголовочный файл wctype.h предлагает аналоги символьных функций из ctype.h наряду с несколькими дополнительными функциями. Кроме того, в нем определены три типа и макрос, как показано в табл. Б.V.51.

Таблица Б.V.51. Типы и макрос, определенные в wctype.h

Тип/макрос	Описание
wint_t	Целочисленный тип, который может содержать любое значение расширенного набора символов плюс как минимум одно значение, не входящее в этот набор
wctrans_t	Скалярный тип, который может представлять локальные специфичные отображения символов
wctype_t	Скалярный тип, который может представлять локальные специфичные классификации символов
WEOF	Константное выражение типа wint_t, которое не соответствует ни одному элементу из расширенного набора символов; эквивалент EOF в широких символах, применяемый для обозначения конца файла при вводе с использованием широких символов

Классификации символов в wctype.h возвращают true (ненулевое значение), если аргумент широкого символа удовлетворяет условиям, описанным функцией. В общем случае функция широких символов возвращает true, если соответствующая функция ctype.h возвращает true для однобайтного символа, который соответствует широкому. Эти функции перечислены в табл. Б.V.52.

Таблица Б.V.52. Функции классификации широких символов

Прототип	Описание
int iswalnum(wint_t wc);	Возвращает true, если wc представляет цифру или букву
int iswalpha(wint_t wc);	Возвращает true, если wc представляет букву
int iswblank(wint_t wc);	Возвращает true, если wc представляет пробел
int iswcntrl(wint_t wc);	Возвращает true, если wc представляет управляющий символ
int iswdigit(wint_t wc);	Возвращает true, если wc представляет десятичную цифру
int iswgraph(wint_t wc);	Возвращает true, если iswprint(wc) равно true, а iswspace(wc) — false
int iswlower(wint_t wc);	Возвращает true, если wc представляет символ нижнего регистра
int iswprint(wint_t wc);	Возвращает true, если wc представляет печатаемый символ
int iswpunct(wint_t wc);	Возвращает true, если wc представляет знак пунктуации
int iswspace(wint_t wc);	Возвращает true, если wc представляет символ табуляции, пробела или новой строки
int iswupper(wint_t wc);	Возвращает true, если wc представляет символ верхнего регистра
int iswxdigit(wint_t wc);	Возвращает true, если wc представляет шестнадцатеричную цифру

Библиотека также содержит две классифицирующие функции, которые называются *расширяемыми*, поскольку для классификации символов они используют значение LC_CTYPE текущей локальной установки. Эти функции перечислены в табл. Б.V.53.

Таблица Б.V.53. Расширяемые функции классификации широких символов

Прототип	Описание
`int iswctype(wint_t wc, wctype_t desc);`	Функция `iswctype()` возвращает `true`, если `wc` обладает свойством, описанным в `desc`. (Ниже приведены пояснения.)
`wctype_t wctype( const char *property);`	Функция `wctype()` конструирует значение типа `wctype_t`, которое описывает класс широких символов, идентифицированный строковым аргументом `property`. Если `property` идентифицирует допустимый класс широких символов в соответствии с категорией `LC_CTYPE` текущей локальной установки, то функция `wctype()` возвращает ненулевое значение, которое подходит в качестве второго аргумента функции `iswctype()`; в противном случае функция возвращает ноль

Допустимые аргументы для wctype() состоят из имен функций классификации широких символов, из которых исключен префикс isw и которые выражены в виде строк. Например, wctype("alpha") характеризует класс символов, проверяемых функцией iswalpha(). Таким образом, вызов

```
iswctype(wc, wctype("alpha"))
```

эквивалентен вызову

```
iswalpha(wc)
```

за исключением того, что символы классифицируются с применением категорий LC_CTYPE.

Заголовочный файл wctype.h предлагает четыре функции преобразования. Две из них являются эквивалентами с широкими символами для функций toupper() и tolower() из библиотеки ctype.h. Третья представляет собой расширенную версию, которая использует локальные настройки LC_CTYPE для определения символов верхнего и нижнего регистра. Четвертая функция предоставляет подходящие классификационные аргументы для третьей функции. Все эти функции перечислены в табл. Б.V.54.

Таблица Б.V.54. Функции трансформации широких символов

Прототип	Описание
`wint_t towlower( wint_t wc);`	Возвращает версию `wc` нижнего регистра, если символ `wc` представлен в верхнем регистре; в противном случае возвращает `wc`
`wint_t towupper( wint_t wc);`	Возвращает версию `wc` верхнего регистра, если символ `wc` представлен в нижнем регистре; в противном случае возвращает `wc`
`wint_t towctrans( wint_t wc, wctrans_t desc);`	Возвращает версию `wc` нижнего регистра (как определено настройками `LC_CTYPE`), если `desc` эквивалентно возвращаемому значению `wctrans("lower")`; возвращает версию `wc` верхнего регистра (как определено настройками `LC_CTYPE`), если `desc` эквивалентно возвращаемому значению `wctrans("upper")`
`wctrans_t wctrans( const char *property);`	Если аргументом является `"upper"` или `"lower"`, функция возвращает значение `wctrans_t`, используемое в качестве аргумента `towctrans()` и отображающее установки `LC_CTYPE`; в противном случае возвращается 0

Раздел VI. Расширенные целочисленные типы

Как было описано в главе 3, заголовочный файл inttypes.h из C99 предоставляет систематизированный набор альтернативных имен для разнообразных целочисленных типов. Эти имена описывают свойства типа более ясно, чем это делают стандартные имена. Например, тип int может быть 16-, 32- или 64-битовым, однако тип int32_t — всегда 32-битовый.

Выражаясь более точно, в заголовочном файле inttypes.h определены макросы, которые могут применяться в функциях scanf() и printf() для чтения и записи целых чисел этих типов. Этот заголовочный файл включает заголовочный файл stdlib.h, который обеспечивает действительные определения типов. Форматирующие макросы — это строки, которые могут быть объединены с другими строками для формирования допустимых директив форматирования.

Типы определены с использованием typedef. Например, в системе с 32-битовым int может применяться такое определение:

```
typedef int int32_t;
```

Спецификаторы формата определены с помощью директив #define. Например, в системе, где используется приведенное выше определение int32_t, могут быть следующие определения:

```
#define PRId32 "d"   // спецификатор вывода
#define SCNd32 "d"   // спецификатор ввода
```

Имея эти определения, вы можете объявлять расширенные целочисленные переменные, вводить их значения и отображать следующим образом:

```
int32_t cd_sales;   // 32-битовое целое число
scanf("%" SCNd32, &cd_sales);
printf("Продажи компакт-дисков составили %10" PRId32 " единиц\n", cd_sales);
```

При необходимости применяется конкатенация строк, чтобы получить финальную управляющую строку. Таким образом, предыдущий код преобразуется к такому виду:

```
int cd_sales;        // 32-битовое целое число
scanf("%d", &cd_sales);
printf("Продажи компакт-дисков составили %10d единиц\n", cd_sales);
```

Если вы перенесете первоначальный код в систему с 16-битовым int, то эта система может определить int32_t как long, PRId32 — как "ld", а SCNd32 — как "ld". Но вы могли бы использовать тот же самый код, зная, что в системе применяется 32-битовый int.

В оставшейся части этого справочного раздела перечислены расширенные типы вместе со спецификаторами формата и макросами, представляющими предельные значения этих типов.

Типы с точной шириной

Один из наборов typedef идентифицирует типы с точными размерами. Общая форма выглядит как int*N*_t для типов со знаком и uint*N*_t — для типов без знака, где *N* обозначает количество битов. Однако обратите внимание, что не все системы способны поддерживать все типы. Например, могла бы существовать система, для которой минимальной используемой единицей памяти является 16 битов; в такой системе не поддерживались бы типы int8_t и uint8_t. Макросы формата могут применять для типов со знаком либо d, либо i, поэтому PRIi8 и SCNi8 также работают.

Для типов без знака вы можете подставлять o, x или X для u, чтобы получить вместо %u спецификаторы %o, %x или %X. Например, вы можете использовать PRIX32, чтобы вывести uint32_t в шестнадцатеричной форме. В табл. Б.VI.1 перечислены типы с точной шириной, спецификаторы формата и предельные значения.

Таблица Б.VI.1. Типы с точной шириной

Имя типа	Спецификатор printf()	Спецификатор scanf()	Минимальное значение	Максимальное значение
int8_t	PRId8	SCNd8	INT8_MIN	INT8_MAX
int16_t	PRId16	SCNd16	INT16_MIN	INT16_MAX
int32_t	PRId32	SCNd32	INT32_MIN	INT32_MAX
int64_t	PRId64	SCNd64	INT64_MIN	INT64_MAX
uint8_t	PRIu8	SCNu8	0	UINT8_MAX
uint16_t	PRIu16	SCNu16	0	UINT16_MAX
uint32_t	PRIu32	SCNu32	0	UINT32_MAX
uint64_t	PRIu64	SCNu64	0	UINT64_MAX

Типы с минимальной шириной

Типы с минимальной шириной гарантируют, что тип имеет размер, равный, как минимум, известному количеству битов. Эти типы существуют всегда. Например, система, которая не поддерживает 8-битовые единицы памяти, могла бы определить int_least_8 как 16-битовый тип. В табл. Б.VI.2 перечислены типы с минимальной шириной, спецификаторы формата и предельные значения.

Таблица Б.VI.2. Типы с минимальной шириной

Имя типа	Спецификатор printf()	Спецификатор scanf()	Минимальное значение	Максимальное значение
int_least8_t	PRILEASTd8	SCNLEASTd8	INT_LEAST8_MIN	INT_LEAST8_MAX
int_least16_t	PRILEASTd16	SCNLEASTd16	INT_LEAST16_MIN	INT_LEAST16_MAX
int_least32_t	PRILEASTd32	SCNLEASTd32	INT_LEAST32_MIN	INT_LEAST32_MAX
int_least64_t	PRILEASTd64	SCNLEASTd64	INT_LEAST64_MIN	INT_LEAST64_MAX
uint_least8_t	PRILEASTu8	SCNLEASTu8	0	UINT_LEAST8_MAX
uint_least16_t	PRILEASTu16	SCNLEASTu16	0	UINT_LEAST16_MAX
uint_least32_t	PRILEASTu32	SCNLEASTu32	0	UINT_LEAST32_MAX
uint_least64_t	PRILEASTu64	SCNLEASTu64	0	UINT_LEAST64_MAX

Самые быстрые типы с минимальной шириной

В отдельной системе некоторые представления целых чисел могут быть быстрее других. Например, int_least16_t может быть реализован как short, но система может выполнять арифметические вычисления быстрее, если используется тип int. Таким образом, в заголовочном файле inttypes.h также определены самые быстрые типы для представления, как минимум, определенного количества битов. Эти типы существуют всегда. В некоторых случаях может отсутствовать очевидный выбор для самого быстрого типа, тогда система просто указывает один из возможных вариантов.

В табл. Б.VI.3 перечислены самые быстрые типы с минимальной шириной, спецификаторы формата и предельные значения.

Таблица Б.VI.3. Самые быстрые типы с минимальной шириной

Имя типа	Спецификатор printf()	Спецификатор scanf()	Минимальное значение	Максимальное значение
int_fast8_t	PRIFASTd8	SCNFASTd8	INT_FAST8_MIN	INT_FAST8_MAX
int_fast16_t	PRIFASTd16	SCNFASTd16	INT_FAST16_MIN	INT_FAST16_MAX
int_fast32_t	PRIFASTd32	SCNFASTd32	INT_FAST32_MIN	INT_FAST32_MAX
int_fast64_t	PRIFASTd64	SCNFASTd64	INT_FAST64_MIN	INT_FAST64_MAX
uint_fast8_t	PRIFASTu8	SCNFASTu8	0	UINT_FAST8_MAX
uint_fast16_t	PRIFASTu16	SCNFASTu16	0	UINT_FAST16_MAX
uint_fast32_t	PRIFASTu32	SCNFASTu32	0	UINT_FAST32_MAX
uint_fast64_t	PRIFASTu64	SCNFASTu64	0	UINT_FAST64_MAX

Типы максимальной ширины

Иногда вам может понадобиться наибольший целочисленный тип из доступных. В табл. Б.VI.4 перечислены такие типы. Фактически они могут быть шире, чем long long или unsigned long long, поскольку система может предоставлять дополнительные типы, более широкие, чем обязательные стандартные типы.

Таблица Б.VI.4. Типы максимальной ширины

Имя типа	Спецификатор printf()	Спецификатор scanf()	Минимальное значение	Максимальное значение
intmax_t	PRIdMAX	SCNdMAX	INTMAX_MIN	INTMAX_MAX
uintmax_t	PRIuMAX	SCNuMAX	0	UINTMAX_MAX

Целые, которые могут хранить указатели

В заголовочном файле inttypes.h (через включенный в него stdint.h) определены два целочисленных типа, приведенные в табл. Б.VI.5, которые могут корректно хранить указатели. То есть, если вы присвоите значение типа void * переменной одного из этих типов, а затем присвоите ее значение обратно указателю, то никакая информация не будет потеряна. Любой из этих типов, или все сразу, могут в системе отсутствовать.

Таблица Б.VI.5. Целочисленные типы, которые могут хранить указатели

Имя типа	Спецификатор printf()	Спецификатор scanf()	Минимальное значение	Максимальное значение
intptr_t	PRIdPTR	SCNdPTR	INTPTR_MIN	INTPTR_MAX
uintptr_t	PRIuPTR	SCNuPTR	0	UINTPTR_MAX

Расширенные целочисленные константы

Вы можете обозначить константу типа long с помощью суффикса L, как в 445566L. А каким образом указать, что константа имеет тип int32_t? Для этого воспользуйтесь макросом, определенным в inttypes.h. Например, выражение INT32_C(445566) расширяется до константы типа int32_t. По существу этот макрос является приведением к лежащему в основе типу, т.е. к фундаментальному типу, представляющему int32_t в конкретной реализации.

Имена макросов формируются из имени типа, в котором _t заменено конструкцией _C и все буквы переведены в верхний регистр. Например, чтобы сделать 1000 константой типа uint_least64_t, примените выражение UINT_LEAST64_C(1000).

Раздел VII. Расширенная поддержка символов

Изначально C не разрабатывался как интернациональный язык программирования. Его набор символов основан на более или менее стандартной клавиатуре, принятой в США. Однако всемирная популярность C привела к появлению ряда расширений, поддерживающих разные и более обширные наборы символов. В этом разделе справочника представлен обзор этих дополнений.

Триграфы

Некоторые клавиатуры содержат не все символы, используемые в C. По этой причине в C предлагаются альтернативные представления некоторых символов с помощью набора трехсимвольных последовательностей, называемых *триграфными последовательностями*, или просто *триграфами*. Такие триграфы перечислены в табл. Б.VII.1.

Таблица Б.VII.1. Триграфы

Триграф	Символ	Триграф	Символ	Триграф	Символ
??=	#	??(	[	??/	\
??)	]	??'	^	??<	{
??!	|	??>	}	??-	~

В файле исходного кода компилятор C заменяет все вхождения триграфов, даже внутри строк в кавычках, соответствующими символами. В результате следующий код:

```
??=include <stdio.h>
??=define LIM 100
int main()
??<
   int q??(LIM??);
   printf("Новости будут скоро.??/n");
   ...
??>
```

становится таким:

```
#include <stdio.h>
#define LIM 100
int main()
{
   int q[LIM];
   printf("Новости будут скоро.\n");
   ...
}
```

Возможно, для активизации этого средства понадобится включить специальный флаг компилятора.

Диграфы

Учитывая громозкость системы триграфов, в стандарте C99 предлагаются двухсимвольные комбинации, называемые *диграфами*, которые могут использоваться вместо ряда стандартных знаков пунктуации C. Эти диграфы перечислены в табл. Б.VII.2.

Таблица Б.VII.2. Диграфы

Диграф	Символ	Диграф	Символ	Диграф	Символ
`<:`	`[`	`:>`	`]`	`<%`	`{`
`%>`	`}`	`%:`	`#`	`%:%:`	`##`

В отличие от триграфов, диграфы внутри строк в кавычках не имеют специального назначения. То есть фрагмент

```
%:include <stdio.h>
%:define LIM 100
int main()
<%
   int q<:LIM:>;
   printf("Новости будут скоро.:>");
   ...
%>
```

ведет себя так же, как и следующий код:

```
#include <stdio.h>
#define LIM 100
int main()
{
   int q[LIM];
 printf("Новости будут скоро.:>");  // :> просто часть строки
   ...
}                                   // :> то же, что и }
```

Альтернативное написание: `iso646.h`

С помощью триграфных последовательностей операцию `||` можно записать как `??!??!`, что не выглядит особенно изящно. Стандарт C99 через заголовочный файл `iso646.h` (см. табл. Б.V.11) предоставляет макросы, которые расширяются в операции. В стандарте эти макросы называются *альтернативным написанием*.

Если вы включите заголовочный файл `iso646.h`, то оператор наподобие

```
if(x == M1 or x == M2)
   x and_eq 0XFF;
```

расширяется следующим образом:

```
if(x == M1 || x == M2)
   x &= 0XFF;
```

Многобайтные символы

Стандарт описывает многобайтный символ как последовательность из одного или более байтов, которая представляет элемент расширенного набора символов либо

в исходной среде, либо в среде выполнения. Исходная среда – это та, где вы подготавливаете исходный код, а среда выполнения – та, в которой вы запускаете скомпилированную программу. Они могут отличаться. Например, программа может разрабатываться в одной среде с намерением запускать ее в другой. Расширенный набор символов является надмножеством базового набора символов, определенного в С.

Реализация может предоставлять расширенный набор символов, который позволяет, к примеру, вводить клавиатурные символы, не соответствующие базовому набору. Они могут применяться в строковых литералах и символьных константах, а также встречаться в файлах. Реализация также может предлагать многобайтные эквиваленты символов из базового набора, которые можно использовать вместо триграфов или диграфов.

Немецкая реализация, например, может разрешить присутствие в строках символов с умляутами:

```
puts("eins zwei drei vier fünf");
```

В общем случае расширенный набор символов, доступный программе, зависит от локальной установки.

Универсальные имена символов (UCN)

Многобайтные символы могут применяться в строках, но не в идентификаторах. Универсальные имена символов (Universal Character Name – UCN) представляют собой дополнение С99, позволяющее использовать символы из расширенного набора в качестве части имен идентификаторов. Система расширяет концепцию управляющих последовательностей для обеспечения возможности кодирования символов из стандарта ISO/IEC 10646. Этот стандарт разработан совместно Международной организацией по стандартизации (ISO) и Международной электротехнической комиссией (IEC), и он предоставляет числовые коды для огромного списка символов. Стандарт 10646 тесно согласован с Unicode (см. врезку ниже).

Unicode и ISO 10646

Unicode предлагает решение по представлению разнообразных наборов символов за счет предоставления стандартной системы нумерации для большого количества символов и знаков, группируя их по типам. Например, кодировка ASCII включена как подмножество Unicode, поэтому символы U.S. Latin, такие как А и Z, имеют одинаковое представление в обеих системах. Но Unicode также содержит в себе другие латинские символы вроде применяемых в европейских языках; символы из других алфавитов, в числе которых кириллический, греческий, иврит, чероки, арабский, тайский и бенгальский; а также идеограммы вроде тех, что используются в китайском и японском языках. К настоящему времени Unicode представляет свыше 110 000 символов и 100 шрифтов и все еще продолжает развиваться. За дополнительными сведениями обращайтесь на веб-сайт Консорциума Unicode по адресу `www.unicode.org`.

Каждому символу в Unicode назначается число, называемое *кодовым знаком*. Типичная форма записи для кодовых знаков Unicode выглядит так: U-222B. Наличие U идентифицирует последовательность как символ Unicode, а 222B — это шестнадцатеричное число для символа, в данном случае знака интеграла.

Международная организация по стандартизации (ISO) организовала рабочую группу для создания ISO 10646, который также является стандартом для кодирования многоязычного текста. Группа ISO 10646 и группа Unicode работают вместе с 1991 года, поддерживая свои стандарты в согласованном состоянии.

Существуют две формы последовательностей UCN. Первая имеет вид \u*hexquad*, где *hexquad* – последовательность из четырех шестнадцатеричных цифр; например, \u00F6. Вторая форма записывается как \U*hexquadhexquad*; например, \U0000AC01. Поскольку каждая шестнадцатеричная цифра соответствует четырем битам, форма \u может применяться для кодов, представимых 16-битовым целым числом, а \U – для кодов, представимых 32-битовым целым числом.

Если в вашей системе реализованы имена UCN и она включает необходимые символы в расширенный набор символов, то имена UCN могут использоваться в строках, символьных константах и идентификаторах:

```
wchar_t value\u00F6\u00F8 = L'\u00f6';
```

Широкие символы

Стандарт C99 посредством библиотек wchar.h и wctype.h обеспечивает еще один вид поддержки для крупных наборов символов посредством применения широких символов. В указанных заголовочных файлах wchar_t определяется как целочисленный тип; точный тип зависит от реализации. Этот тип предназначен для хранения символов из расширенного набора символов, который является надмножеством базового набора символов. По определению типа char достаточно для работы с базовым набором символов. Типу wchar_t может потребоваться больше битов для обработки более широкого диапазона значений кодов. Например, char может быть 8-битовым байтом, а wchar_t – 16-битовым unsigned short.

Константы широких символов и строковые литералы обозначаются префиксом L, а для отображения данных с широкими символами можно использовать модификаторы %lc и %ls:

```
wchar_t wch = L'Я';
wchar_t w_arr[20] = L"являюсь широким!";
printf("%lc %ls\n", wch, w_arr);
```

Если, к примеру, wchar_t реализован как 2-байтовая единица, то однобайтный код 'Я' должен быть сохранен в младшем байте переменной wch. Символы, не входящие в стандартный набор, могут потребовать обоих байтов для размещения своего кода. Вы можете применять универсальные коды символов для обозначения символов, значения кодов которых выходят за пределы диапазона char:

```
wchar_t w = L'\u00E2'; /* 16-битовое значение кода */
```

Массив значений wchar_t может содержать строку широких символов, причем каждый его элемент будет хранить код одиночного широкого символа. Значение wchar_t со значением кода 0 является эквивалентом wchar_t нулевого символа и называется *нулевым широким символом*. Он используется для завершения строк широких символов.

Для чтения широких символов можно применять спецификаторы %lc и %ls:

```
wchar_t wch1;
wchar_t w_arr[20];
puts("Введите свою научную степень:");
scanf("%lc", &wch1);
puts("Введите свою фамилию:");
scanf("%ls",w_arr);
```

Заголовочный файл wchar.h предоставляет дополнительную поддержку широких символов. В частности, он предлагает функции ввода-вывода широких символов, функции преобразования широких символов и функции манипуляции строками.

По большей части они являются эквивалентами существующих функций, но просто имеют дело с широкими символами. Например, вы можете использовать fwprintf() и wprintf() для вывода, а fwscanf() и wscanf() — для ввода. Главное отличие заключается в том, что эти функции требуют управляющей строки широких символов и работают с потоками ввода и вывода широких символов. Например, следующий фрагмент отображает информацию в виде последовательность широких символов:

```
wchar_t * pw = L"Указывает на строку широких символов";
int dozen = 12;
wprintf(L"Элемент %d: %ls\n", dozen, pw);
```

Подобным же образом существуют функции getwchar(), putwchar(), fgetws() и fputws(). В wchar.h определен макрос WEOF, который играет ту же роль, что и EOF для ввода-вывода, ориентированного на байты. Макрос WEOF должен быть значением, которое не соответствует ни одному допустимому символу. Поскольку возможно, что все значения типа wchar_t представляют собой допустимые символы, в библиотеке определен тип wint_t, который охватывает все возможные значения wchar_t плюс WEOF.

Имеются эквиваленты для функций из библиотеки string.h. Например, wcscpy(ws2, ws1) копирует строку широких символов, указанную посредством ws1, в массив широких символов, на который указывает ws2. Аналогично, для сравнения двух широких строк предусмотрена функция wcscmp() и т.д.

Заголовочный файл wctype.h добавляет к общему набору функции классификации символов. Например, iswdigit() возвращает true, если ее аргумент в виде широкого символа является цифрой, а iswblank() возвращает true, если ее аргумент представляет собой пробельный символ. Стандартными значениями для пробельного символа являются пробел, записанный как L' ', и горизонтальная табуляция, записанная как L'\t'.

Стандарт C11 через заголовочные файлы uchar.h предоставляет дополнительную поддержку широких символов за счет определения двух типов, предназначенных для соответствия двум распространенным форматам кодирования Unicode. Первый тип, char16_t, представляет собой наименьший доступный целочисленный тип без знака, который способен хранить 16-битовый код. Он может применяться с формой UCN, содержащей четыре шестнадцатеричные цифры, и схемой кодировки UTF-16 в Unicode:

```
char16_t = '\u00F6';
```

Второй тип, char32_t, определяет наименьший доступный целочисленный тип без знака, способный хранить 32-битовый код. Он может использоваться с формой UCN, содержащей восемь шестнадцатеричных цифры, и схемой кодировки UTF-32 в Unicode:

```
char32_t = '\U0000AC01';
```

Для обозначения строк char16_t и char32_t можно применять префиксы u и U соответственно:

```
char16_t ws16[11] = u"Tannh\u00E4user";
char32_t ws32[13] = U"caf\U000000E9 au lait";
```

Обратите внимание, что эти два типа более специфичны, чем wchar_t. Например, тип wchar_t может быть достаточно широк для хранения 32-битовых кодов в одной системе, но иметь ширину для сохранения только 16-битовых кодов в другой системе. Кроме того, два новых типа совместимы с синтаксисом C++.

Широкие и многобайтные символы

Широкие и многобайтные символы – это два разных подхода к обработке расширенных наборов символов. Многобайтный символ может быть представлен одним, двумя, тремя и более байтами. Все широкие символы будут иметь одинаковую ширину. Многобайтные символы могут использовать состояние сдвига (т.е. байт, который определяет, как должны интерпретироваться последующие байты); широкие символы не поддерживают состояние сдвига. Файл многобайтных символов может быть прочитан в обычный массив char с помощью стандартных функций ввода; файл широких символов должен быть прочитан в массив широких символов с применением одной из функций ввода широких символов.

Стандарт C99 через библиотеку wchar.h предоставляет функции, предназначенные для выполнения преобразований между многобайтным представлением и представлением wchar_t. Функция mbrtowc() преобразует многобайтный символ в широкий, а функция wcrtomb() – широкий символ в многобайтный. Подобным же образом функция mbstrtowcs() преобразует многобайтную строку в строку широких символов, а функция wcstrtombs() – строку широких символов в многобайтную.

Стандарт C11 посредством библиотеки uchar.h предоставляет функции для преобразования между многобайтным представлением и представлением char16_t, а также между многобайтным представлением и представлением char32_t.

Раздел VIII. Расширенные вычислительные средства C99/C11

Исторически сложилось так, что первым языком для выполнения числовых научных и инженерных вычислений был FORTRAN. Стандарт C90 привел методы вычислений C в более полное соответствие с FORTRAN. Например, спецификация характеристик плавающей запятой, использованная в float.h, основана на модели, которая разработана комитетом по стандартизации FORTRAN. В стандартах C99 и C11 продолжена работа по расширению пригодности языка C к решению вычислительных задач. Например, массивы переменной длины, добавленные в C99, но сделанные необязательными в C11, в большей степени соответствуют стилю применения массивов FORTRAN, чем традиционные массивы C. (В C11 определен макрос __STDC_NO_VLA__, который развертывается в 1, если реализация не поддерживает массивы переменной длины.)

Стандарт плавающей запятой IEC

Международная электротехническая комиссия (IEC) опубликовала стандарт вычислений с плавающей запятой (IEC 60559). Этот стандарт включает описание форматов плавающей запятой, точности, представления NaN, бесконечности, методик округления, преобразований, исключений, рекомендованных функций и алгоритмов и тому подобного. В C99 этот стандарт был принят как руководство по реализации вычислений с плавающей запятой на языке C. Большая часть дополнений C99, касающихся средств работы с плавающей запятой, являются частью этих усилий. Речь идет о таких вещах, как заголовочный файл fenv.h и некоторые из новых математических функций. Кроме того, в заголовочном файле float.h определено несколько макросов, имеющих отношение к модели плавающей запятой IEC.

Модель плавающей запятой

Давайте вкратце рассмотрим модель плавающей запятой. Стандарт представляет значение с плавающей запятой *x* как степень основания системы счисления, умноженная на дробную часть, которая представлена в этой системе счисления, в отличие от экспоненциальной формы записи языка C, где мы можем записать 876.54 в виде 0.87654E3. Как и можно было ожидать, формальное представление выглядит более внушительно:

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}$$

Выражаясь кратко, эта формула представляет число как результат умножения показателя степени, или экспоненты (e), основания системы счисления (b) на *значащую часть числа*, т.е. многозначную дробную часть.

Ниже описаны различные компоненты формального представления.

- s — знак (±1).
- b — это *основание системы счисления.* Наиболее распространенным значением является 2, т.к. в процессорах с плавающей запятой обычно применяется двоичная арифметика.
- e представляет целочисленную экспоненту. (Ее не следует путать с числовой константой e, используемой в качестве основания для натуральных логарифмов.) Экспонента будет ограничена диапазоном, имеющим минимальное и максимальное значение. Эти значения будут зависеть от количества битов, выделенных для хранения экспоненты.
- f_k представляет возможные цифры для основания системы счисления b; например, при основании 2 возможными цифрами являются 0 и 1, а при основании 16 — цифры из диапазона от 0 до F.
- p — это точность, т.е. количество цифр по основанию b, применяемых для представления значащей части числа. Значение p будет ограничено количеством битов, выделенных для хранения значащей части числа.

Понимание этого представления является ключом к пониманию содержимого `float.h` и `fenv.h`, поэтому давайте ознакомимся с двумя примерами, иллюстрирующими работу представления плавающей запятой.

Прежде всего, предположим, что основанием системы счисления b является 10, а точность p составляет 5. Тогда значение 24.51 можно было бы записать следующим образом:

$$(+1)10^3(2/10 + 4/100 + 5/1000 + 1/10000 + 0/100000)$$

Полагая, что компьютер способен хранить десятичные цифры, он мог бы сохранить знак, экспоненту 3, а также пять значений f_k — 2, 4, 5, 1 и 0. (Здесь f_1 — это 2, f_2 — 4 и т.д.) Таким образом, значащей частью является 0.24510. Умножение ее на 10^3 дает 24.51.

Далее предположим, как более распространенную ситуацию, что основание системы счисления b представляет собой 2. Пусть p равно 7, экспонента равна 5, а значащая часть числа хранится как 1011001, используя 7 двоичных цифр, как описано посредством p. Будем считать, что знак числа положителен.

Тогда согласно формуле мы можем построить число так:

$x = (+1)2^5(1/2 + 0/4 + 1/8 + 1/16 + 0/32 + 0/64 + 1/128)$
$= 32(1/2 + 0/4 + 1/8 + 1/16 + 0/32 + 0/64 + 1/128)$
$= 16 + 0 + 4 + 2 + 0 + 0 + 1/4 = 22.25$

Многие макросы из `float.h` имеют отношение к этому представлению. Например, для значения типа `float` макрос `FLT_RADIX` – это *b*, т.е. применяемое основание системы счисления, а `FLT_MANT_DIG` – это *p*, т.е. количество цифр (по основанию *b*) в значащей части числа.

Нормальные и субнормальные значения

Концепция *нормализованного значения с плавающей запятой* играет важную роль, поэтому давайте изучим ее. Для простоты предположим, что основанием системы счисления является 10 (*b* = `FLT_RADIX` = 10), а в значащей части числа используется 5 десятичных цифр для значения типа `float` (*p* = `FLT_MANT_DIG` = 5). (Стандарт требует более высокой точности, чем эта, но желание упростить изучение позволяет нам проигнорировать данное требование.) Взгляните на следующие способы представления значения 31.841:

экспонента = 3, значащая часть = .31841 (.31841E3)

экспонента = 4, значащая часть = .03184 (.03184E4)

экспонента = 5, значащая часть = .00318 (.00318E5)

Очевидно, что первый метод будет наиболее точным, поскольку в значащей части присутствуют все пять доступных цифр. Нормализованное ненулевое значение с плавающей запятой – это такое значение, в котором первая цифра в значащей части отличается от нуля, и именно так обычно хранятся значения с плавающей запятой.

Теперь предположим, что минимальная экспонента (`FLT_MIN_EXP`) установлена в -10. Тогда наименьшее нормализованное значение таково:

экспонента = -10, значащая часть = .10000 (.10000E-10)

Обычно умножение или деление на 10 означает увеличение или уменьшение экспоненты, но в этом случае деление не может дополнительно уменьшить экспоненту. Однако можно изменить значащую часть, чтобы получить следующее представление:

экспонента = -10, значащая часть = 0.0100 (.01000E-10)

Такое число будет называться субнормальным, потому что оно не использует полную точность значащей части. Например, деление 0.12343E-10 на 10 дает .01234E-10, и одна цифра теряется.

В данном примере 0.1000E-10 является наименьшим ненулевым нормальным представлением (`FLT_MIN`), а наименьшим ненулевым субнормальным значением будет 0.00001E-10 (`FLT_TRUE_MIN`).

Макросы `FLT_HAS_SUBNORM`, `DBL_HAS_SUBNORM` и `LDBL_HAS_SUBNORM` из `float.h` характеризуют, каким образом реализация обрабатывает субнормальные значения. Ниже перечислены возможные значения для этих макросов и объяснен их смысл:

- `-1` неопределимо (непротиворечивой интерпретации не существует)
- `0` отсутствует (реализация могла бы, например, заменять субнормальные значения нулем)
- `1` присутствует

Библиотека math.h предлагает средства, в том числе макросы fpclassify() и isnormal(), которые позволяют идентифицировать, когда программа генерирует субнормальные значения, таким образом, теряя точность.

Схемы оценки

Макрос FLT_EVAL_METHOD из float.h отражает схему, которую реализация применяет для оценки выражений с плавающей запятой. Возможные варианты описаны ниже:

- -1 неопределимо
- 0 оценка операций и констант в диапазоне и точности их типа
- 1 оценка операций и констант типов float и double в диапазоне и точности типа double, а операций и констант типа long double в диапазоне и точности типа long double
- 2 оценка операций и констант всех типов с плавающей запятой в диапазоне и точности типа long double

Реализации разрешено предоставлять дополнительные отрицательные значения для обозначения других вариантов.

Предположим для примера, что программа умножает два значения float, присваивая результат переменной float. При варианте 1, который был выбран в K&R C, два значения float расширялись до double, вычисление производилось с использованием double, а результат округлялся до float при присваивании.

При варианте 0, который выбран в ANSI C, два значения float умножаются как float и затем выполняется присваивание результата. Это может быть более быстрой операцией, чем вариант 1, но здесь есть вероятность небольшой потери точности.

Округление

Макрос FLT_ROUNDS из float.h описывает то, каким образом система обрабатывает округление. Ниже перечислены возможные варианты округления:

- -1 неопределимо
- 0 в сторону нуля
- 1 к ближайшему значению
- 2 в сторону положительной бесконечности
- 3 в сторону отрицательной бесконечности

В системе с другими вариантами могут быть определены дополнительные значения.

В некоторых системах предлагается возможность управления схемой округления, и в таком случае функция fesetround() из fenv.h предоставляет программный контроль.

Побочные эффекты разных методов округления, вероятно, не будут особенно важны, если вы подсчитываете объем муки для 37 пирожных, но они могут повлиять, скажем, в случае проведения важных финансовых и научных вычислений. Понятно, что метод округления вступает в игру, когда выполняется преобразование значения с плавающей запятой высокой точности в значение меньшей точности, например, во время присваивания результата вычисления double переменной float. Метод округления может быть также задействован при изменении основания системы счисления. Дробная часть с точным представлением в одной системе счисления может не иметь точного представления в другой системе счисления.

Взгляните на следующий код:

```
float x = 0.8;
```

Дробь 8/10, или 4/5, может быть точно представлена в системе по основанию 10. Но большинство компьютерных систем будут хранить результат по основанию 2, а в двоичной системе счисления 4/5 дает бесконечную дробь:

0.1100110011001100...

Таким образом, при сохранении в x величина 0.8 округляется до приближенного значения, которое может зависеть от метода округления.

Однако может случиться так, что конкретная реализация не удовлетворяет требованиям IEC 60559; например, из-за того, что имеющееся оборудование на подобное не рассчитано. По этой причине в C99 определены два макроса, которые могут применяться в директивах препроцессора для проверки соответствия. Во-первых, макрос

```
__STDC_IEC_559__
```

условно определен как константа 1, если реализация отвечает спецификациям плавающей запятой IEC 60559. Во-вторых, макрос

```
__STDC_IEC_559_COMPLEX__
```

условно определен как константа 1, если реализация придерживается совместимой с IEC 60559 арифметики комплексных чисел.

Если в реализации эти макросы не определены, значит, нет никакой гарантии совместимости с IEC 60559.

Заголовочный файл fenv.h

Заголовочный файл fenv.h предоставляет средство взаимодействия со средой плавающей запятой. Другими словами, он позволяет устанавливать *значения управляющего режима* плавающей запятой, который определяет порядок выполнения вычислений с плавающей запятой, а также позволяет выяснять значения флагов состояния плавающей запятой, или *исключения*, которые сообщают информацию о результатах арифметических вычислений. Примером настроек управляющего режима может служить способ округления чисел. Примером флага состояния может быть флаг, устанавливаемый операциями, которые вызывают переполнение с плавающей запятой. Операция, устанавливающая флаг состояния, описывается как *генерирующая исключение*.

Флаги состояния и управляющие режимы имеют смысл, только если их поддерживает оборудование. Например, вы не можете изменить метод округления, если оборудование не позволяет этого делать.

Чтобы включить поддержку режимов и флагов, используется следующая директива препроцессора:

```
#pragma STDC FENV_ACCESS ON
```

Поддержка остается включенной до тех пор, пока программа не достигнет конца блока, содержащего прагму, либо, если прагма является внешней, то до конца файла или единицы трансляции. Для отключения поддержки применяется такая директива:

```
#pragma STDC FENV_ACCESS OFF
```

Можно также записать следующую прагму:

```
#pragma STDC FENV_ACCESS DEFAULT
```

Это восстановит стандартное состояние компилятора, которое зависит от реализации.

Данное средство важно для тех разработчиков, которые имеют дело с критически важными вычислениями с плавающей запятой, но представляет ограниченный интерес для большинства пользователей, поэтому здесь мы не будем вдаваться в особые детали.

Прагма STDC FP_CONTRACT

Некоторые процессоры с плавающей запятой могут объединять выражения с плавающей запятой, содержащие множество операций, в единую операцию. Например, процессор может быть в состоянии выполнить оценку следующего выражения за один шаг:

```
x*y - z
```

Это увеличивает скорость вычислений, но может привести к снижению их предсказуемости. Прагма `STDC FP_CONTRACT` позволяет включать и отключать данное средство. Стандартное состояние зависит от реализации.

Чтобы отключить это средство сжатия для определенного вычисления, а затем включить его снова, можно поступить так:

```
#pragma STDC FP_CONTRACT OFF
val = x * y - z;
#pragma STDC FP_CONTRACT ON
```

Дополнения библиотеки math.h

В математической библиотеке С90 функции по большей части объявлены с аргументами `double` и типом возвращаемого значения `double`:

```
double sin(double);
double sqrt(double);
```

Библиотеки С99 и С11 предоставляют версии `float` и `long double` для всех этих функций. Имена такие функций содержат суффиксы `f` и `l`:

```
float sinf(float);                /* версия float функции sin()       */
long double sinl(long double);  /* версия long double функции sin() */
```

Наличие семейств функций с разными уровнями точности позволяет выбирать наиболее эффективную комбинацию типов и функций, необходимую для достижения конкретной цели.

В С99 также добавлено несколько функций, часто используемых в научных, инженерных и математических вычислениях. Такие дополнения С99 продемонстрированы в табл. Б.V.16, где перечислены версии `double` всех математических функций. Во многих случаях эти функции возвращают значения, которые могут быть вычислены существующими функциями, однако новые функции делают это быстрее или с большей точностью. Например, `log1p(x)` представляет то же значение, что и `log(1 + x)`, но в `log1p(x)` применяется другой алгоритм, который обеспечивает более точный результат при малых значениях x. Поэтому вы должны использовать функцию `log()` для вычислений в большинстве случаев, а `log1p(x)` — в случаях с малыми значениями x, когда важна высокая точность.

В дополнение к этим функциям в математической библиотеке определено несколько констант и функций, связанных с классификацией чисел и их округлением. Например, значение может быть классифицировано как бесконечное, не число (`NaN`), нормальное, субнормальное и истинный ноль. (`NaN` — это специальное значение, указывающее на то, что значение не является числом; например, `asin(2.0)` возвращает `NaN`,

т.к. asin() может принимать аргументы от -1 до 1. Субнормальным называется такое число, абсолютная величина которого меньше минимально допустимого значения, которое можно представить с максимальной точностью.) Существуют также специализированные функции сравнения, которые ведут себя иначе, чем стандартные операции отношений, когда один или более аргументов являются ненормальными значениями.

Схемы классификации C99 можно применять для обнаружения разного рода отклонений от нормы в вычислениях. Например, макрос isnormal() из math.h возвращает true, если его аргумент является нормальным числом. Ниже представлен пример кода, в котором эта функция используется для прекращения цикла, когда число становится субнормальным:

```
#include <math.h>       // для isnormal()
...
float num = 1.7e-19;
float numprev = num;
while (isnormal(num))  // пока num имеет полную точность float
{
   numprev = num;
   num /= 13.7f;
}
```

Короче говоря, существует расширенная поддержка детализированного управления выполнением вычислений с плавающей запятой.

Поддержка комплексных чисел

Комплексное число — это число, состоящее из действительной и мнимой частей. Действительная часть является обычным вещественным числом, которое может быть представлено типом с плавающей запятой. Мнимая часть представляет мнимое число. Мнимое число, в свою очередь, представляет собой величину, кратную квадратному корню из -1. В математике комплексные числа часто записываются в форме 4.2 + 2.0*i*; где *i* символически представляет квадратный корень из -1.

В C99 поддерживаются три комплексных типа:

- float _Complex
- double _Complex
- long double _Complex

Скажем, значение float _Complex должно сохраняться с применением того же самого способа организации памяти, что и для двухэлементного массива float, где действительная часть сохраняется в первом элементе, а мнимая — во втором.

Реализации C99 и C11 могут поддерживать три мнимых типа:

- float _Imaginary
- double _Imaginary
- long double _Imaginary

Включение заголовочного файла complex.h позволяет использовать имя complex вместо _Complex и imaginary вместо _Imaginary.

Для комплексных типов определены арифметические операции, следующие обычным правилам математики. Например, значение (a+b*I)*(c+d*I) равно:

```
(a*c-b*d)+(b*c+a*d)*I
```

В заголовочном файле complex.h определено несколько макросов и функций, которые принимают и возвращают комплексные числа. В частности, макрос I представляет квадратный корень из −1. Это позволяет поступать следующим образом:

```
double complex c1 = 4.2 + 2.0 * I;
float imaginary c2= -3.0 * I;
```

Для присваивания значений комплексному числу в C11 предлагается второе средство — макрос CMPLX(). Например, если re и im являются значениями double, можно сделать так:

```
double complex c3 = CMPLX(re, im);
```

Смысл в том, что этот макрос может лучше обрабатывать необычные случаи, такие как ситуация, в которой im представляет собой бесконечность или не число, чем прямое присваивание.

В заголовочном файле complex.h содержится несколько прототипов комплексных функций. Многие из них являются комплексными эквивалентами функций math.h, в именах которых присутствует префикс c. Скажем, csin() возвращает комплексный синус своего комплексного аргумента. Другие функции касаются специфических свойств комплексных чисел. Например, creal() возвращает действительную часть комплексного числа, а cimag() — его мнимую часть в виде вещественного числа. То есть для переменной z типа double complex справедливо следующее:

```
z = creal(z) + cimag(z) * I;
```

Если вы знакомы с комплексными числами и нуждаетесь в их применении, вам стоит внимательно исследовать заголовочный файл complex.h.

Ниже показана короткая программа, иллюстрирующая часть поддержки комплексных чисел.

```
// complex.c -- комплексные числа
#include <stdio.h>
#include <complex.h>
void show_cmlx(complex double cv);
int main(void)
{
    complex double v1 = 4.0 + 3.0*I;
    double re, im;
    complex double v2;
    complex double sum, prod, conjug;

    printf("Введите действительную часть комплексного числа: ");
    scanf("%lf", &re);
    printf("Введите мнимую часть комплексного числа: ");
    scanf("%lf", &im);
    // CMPLX() - это средство C11
    // v2 = CMPLX(re, im);
    v2 = re + im * I;
    printf("v1: ");
    show_cmlx(v1);
    putchar('\n');
    printf("v2: ");
    show_cmlx(v2);
    putchar('\n');
    sum = v1 + v2;
    prod = v1 * v2;
    conjug =conj(v1);
```

```
    printf("сумма: ");
    show_cmlx(sum);
    putchar('\n');
    printf("произведение: ");
    show_cmlx(prod);
    putchar('\n');
    printf("комплексное сопряжение v1: ");
    show_cmlx(conjug);
    putchar('\n');

    return 0;
}

void show_cmlx(complex double cv)
{
    printf("(%.2f, %.2fi)", creal(cv), cimag(cv));
    return;
}
```

Если вы используете C++, то вам следует иметь в виду, что заголовочный файл C++ по имени complex предлагает способ работы с комплексными числами, основанный на классах, который существенно отличается от средств complex.h языка C.

Раздел IX. Отличия между C и C++

Большей частью язык C++ представляет собой надмножество C — в том смысле, что допустимая программа на C также является допустимой программой на C++. Главные отличия между C++ и C связаны со многими дополнительными средствами, которые поддерживает язык C++. Тем не менее, существует ряд областей, где правила C++ несколько отличаются от их эквивалентов в C. Эти отличия могут стать причиной того, что программа на C будет работать слегка иначе или даже вовсе не работать, если вы скомпилируете ее как программу C++. Именно эти отличия рассматриваются в данном разделе приложения. Если вы компилируете свои программы на C компилятором, который поддерживает только C++, но не C, вам необходимо знать о таких отличиях. Хотя они весьма незначительно влияют на примеры, приведенные в книге, иногда отличия могут приводить к тому, что некоторые экземпляры допустимого кода C вызывают появление сообщений об ошибках, если код компилируется как программа C++.

Выпуск стандарта C99 усложняет ситуацию, поскольку в некоторых местах он делает C ближе к C++. Например, он позволяет разносить объявления по телу кода и распознает вариант комментария //. В других отношениях C99 углубляет отличия от C++, например, за счет добавления массивов переменной длины и ключевого слова restrict. Стандарт C11 кое в чем уменьшил брешь, к примеру, введя тип char16_t, добавив ключевое слово _Alignas и создав макрос alignas для соответствия ключевому слову C++. Учитывая раннюю стадию развития C11 и неполное принятие C99 некоторыми поставщиками, мы сталкиваемся с отличиями между C90, C99 и C11, а также с отличиями между C++11 и каждым из упомянутых стандартов C. В этом разделе мы обратимся к будущему и обсудим ряд отличий между C99, C11 и C++. Тем временем, язык C++ также развивается, так что точные соответствия и несовпадения между языками C и C++ продолжат изменяться.

Прототипы функций

В отличие от C, в языке C++ прототипирование функций является обязательным. Это отличие проявляется, когда вы оставляете пустыми скобки в объявлении функции.

В языке С пустые скобки означают, что это заблаговременное прототипирование, но в С++ это значит, что функция не имеет параметров. Таким образом, в С++ прототип

```
int slice();
```

означает то же самое, что и

```
int slice(void);
```

Например, следующая последовательность приемлема в устаревшем С, но считается ошибочной с С++:

```
int slice();
int main()
{
   ...
   slice(20, 50);
   ...
}
int slice(int a, int b)
{

}
```

В С компилятор предполагает, что вы применяете старую форму объявления функций. В С++ компилятор предполагает, что `slice()` — это то же самое, что `slice(void)`, и вам не удастся объявить функцию `slice(int, int)`.

К тому же язык С++ позволяет объявлять более одной функции с одним и тем же именем при условии, что они имеют разные списки аргументов.

Константы char

В С константы `char` трактуются как имеющие тип `int`, а в С++ — как принадлежащие типу `char`. Например, рассмотрим следующий оператор:

```
char ch = 'A';
```

В языке С константа `'A'` сохраняется в области памяти с размером `int`; точнее говоря, код этого символа сохраняется как значение типа `int`. То же самое числовое значение также сохраняется в переменной `ch`, но здесь оно занимает только один байт памяти.

С другой стороны, в С++ для `'A'` используется один байт, как и для `ch`. Это отличие не касается каких-либо примеров, рассмотренных в книге. Тем не менее, в некоторых программах на С применяются константы `char`, имеющие тип `int`, за счет использования символьной формы записи для представления целочисленных значений. Например, если система поддерживает 4-байтовый `int`, в С можно поступить следующим образом:

```
int x = 'ABCD';  /* нормально для 4-байтового int в С, но не в С++ */
```

Смысл `'ABCD'` заключается в том, что это 4-байтовое значение `int`, в котором первый байт хранит символьный код буквы *A*, второй — символьный код *B* и т.д. Обратите внимание, что `'ABCD'` существенно отличается от `"ABCD"`. Первое значение — это просто причудливый способ написания значения `int`, но второе — это строка, которая соответствует адресу 5-байтового участка памяти.

Рассмотрим следующий код:

```
int x = 'ABCD';
char c = 'ABCD';
printf("%d %d %c %c\n", x, 'ABCD', c, 'ABCD');
```

В нашей системе этот код генерирует такой вывод:

```
1094861636 1094861636 D D
```

Данный пример иллюстрирует, что если трактовать 'ABCD' как int, то это будет 4-байтовое целое значение, но если трактовать его как char, то программа обращает внимание только на последний байт. Попытка вывода 'ABCD' со спецификатором формата %s в нашей системе приводит к аварийному завершению программы, потому что числовое значение 'ABCD' (1094861636) выходит за рамки допустимых адресов.

Смысл применения значений, подобных 'ABCD', заключается в том, что так можно установить каждый байт в int независимым образом, поскольку каждый символ в точности соответствует одному байту. Однако более удачный подход, который не зависит от конкретных кодов символов, предусматривает использование для целочисленных констант шестнадцатеричных значений, учитывая тот факт, что каждая пара шестнадцатеричных цифр соответствует одному байту. Этот прием обсуждался в главе 15. (Ранние версии C не предоставляли шестнадцатеричной системы записи, так что, по всей видимости, сначала был разработан прием с многосимвольными константами.)

Модификатор const

В C глобальные идентификаторы const имеют внешнее связывание, но в C++ — внутреннее связывание. Другими словами, объявление в C++

```
const double PI = 3.14159;
```

эквивалентно следующему объявлению в C:

```
static const double PI = 3.14159;
```

при условии, что оба они находятся вне функций. Правила C++ нацелены на то, чтобы упростить применение const в файлах заголовков. Если константа имеет внутреннее связывание, то каждый файл, включающий заголовок, получает собственную копию константы. Если же константа имеет внешнее связывание, то в одном файле должно присутствовать определяющее объявление, а во всех остальных файлах должно использоваться ссылочное объявление с ключевым словом extern.

Кстати, в C++ ключевое слово extern может применяться для обеспечения внешнего связывания значения const, поэтому оба языка позволяют создавать константы с внешним и внутренним связыванием. Разница состоит лишь в том, какой вид связывания используется по умолчанию.

Одно дополнительное свойство ключевого слова const в C++ заключается в том, что оно может применяться для объявления размера обычного массива:

```
const int ARSIZE = 100;
double loons[ARSIZE];  /* в C++ это то же, что и double loons[100]; */
```

Такое же объявление можно делать и в C99, но это приведет к созданию массива переменной длины.

В языке C++, но не в C, значения const можно использовать для инициализации других значений const:

```
const double RATE = 0.06;           // допустимо в C++, C
const double STEP = 24.5;           // допустимо в C++, C
const double LEVEL = RATE * STEP;   // допустимо в C++, недопустимо в C
```

Структуры и объединения

После объявления структуры или объединения с дескриптором в C++ этот дескриптор можно применять в качестве имени типа:

```
struct duo
{
   int a;
   int b;
};
struct duo m;  /* допустимо в C, C++              */
duo n;         /* недопустимо в C, допустимо в C++ */
```

В результате имя структуры может конфликтовать с именем переменной. Например, следующая программа компилируется как программа на C, но не может быть успешно скомпилирована как программа на C++, поскольку C++ интерпретирует duo в операторе printf() как тип структуры, а не как внешнюю переменную:

```
#include <stdio.h>
float duo = 100.3;
int main(void)
{
   struct duo { int a; int b;};
   struct duo y = { 2, 4};
   printf ("%f\n", duo);   /* допустимо в C, но не в C++ */
   return 0;
}
```

В C и C++ можно объявлять одну структуру внутри другой:

```
struct box
{
   struct point {int x; int y; } upperleft;
   struct point lowerright;
};
```

В языке C любую из структур можно использовать позже, но C++ требует специальной формы записи для вложенной структуры:

```
struct box ad;          /* допустимо в C, C++              */
struct point dot;       /* допустимо в C, недопустимо в C++ */
box::point dot;         /* недопустимо в C, допустимо в C++ */
```

Перечисления

Язык C++ строже C в отношении применения перечислений. В частности, практически единственное, что можно делать с переменной enum — это присваивать ей константу enum и сравнивать ее с другими значениями. Нельзя присваивать значения int переменной enum без явного приведения типа, равно как не допускается инкрементировать переменную enum. Следующий код иллюстрирует эти утверждения:

```
enum sample {sage, thyme, salt, pepper};
enum sample season;
season = sage;          /* допустимо в C, C++              */
season = 2;             /* предупреждение в C, ошибка в C++ */
season = (enum sample) 3; /* допустимо в C, C++              */
season++;               /* допустимо в C, ошибка в C++      */
```

Кроме того, C++ позволяет не указывать ключевое слово enum при объявлении переменной:

```
enum sample {sage, thyme, salt, pepper};
sample season;  /* недопустимо в C, допустимо в C++ */
```

Как в случае со структурами и объединениями, это может привести к конфликтам, если имена переменной и типа enum совпадают.

Указатель на void

В языке C++, как и в C, указателю на void можно присвоить указатель любого типа, но в отличие от C, указатель на void нельзя присваивать указателю другого типа без явного приведения. Сказанное демонстрируется в следующем коде:

```
int ar[5] = {4, 5, 6,7, 8};
int * pi;
void * pv;
pv = ar;           /* допустимо в C, C++              */
pi = pv;           /* допустимо в C, недопустимо в C++ */
pi = (int * ) pv;   /* допустимо в C, C++              */
```

Еще одно отличие C++ заключается в том, что вы можете присваивать адрес объекта производного класса указателю на базовый класс, но это относится к возможностям, которые в C отсутствуют.

Булевские типы

В языке C++ булевский тип называется bool, а true и false являются ключевыми словами. В языке C булевский тип имеет название _Bool, но включение заголовочного файла stdbool.h делает доступными bool, true и false.

Альтернативное написание

В C++ альтернативное написание or для || и прочих операций обеспечивается ключевыми словами. В C99 и C11 альтернативные написания определены в виде макросов, и для того, чтобы сделать их доступными, необходимо включить заголовочный файл iso646.h.

Поддержка широких символов

В C++ тип wchar_t является встроенным, а wchar_t — ключевым словом. В C99 и C11 тип wchar_t определен в нескольких заголовочных файлах (stddef.h, stdlib.h, wchar.h, wctype.h). Подобным же образом, char16_t и char32_t – это ключевые слова в C++11, но макросы, определенные в uchar.h, в C11.

Язык C++ предоставляет поддержку ввода-вывода широких символов (wchar_t, char16_t и char32_t) через заголовочный файл iostream, тогда как C99 предлагает совершенно другой пакет поддержки ввода-вывода, доступный через заголовочный файл wchar.h.

Комплексные типы

Язык C++ поддерживает комплексные типы посредством класса complex, предоставляемого с помощью заголовочного файла complex. Язык C имеет встроенные комплексные типы и поддерживает их через заголовочный файл complex.h. Эти два подхода существенно отличаются и несовместимы друг с другом. Версия C в большей степени отражает практические потребности вычислительного сообщества.

Встраиваемые функции

В стандарте C99 была добавлена поддержка встраиваемых функций — средства, которое давно существовало в языке C++. Однако их реализация в C99 является более гибкой. В C++ встраиваемая функция по умолчанию имеет внутреннее связывание. Если встраиваемая функция C++ появляется в более чем одном файле, то она должна иметь одно и то же определение, используя те же самые лексемы. Например, один файл не может иметь определение с параметром типа `int`, а другой — определение с параметром типа `int32_t`, несмотря на то, что `int32_t` — это `typedef` для `int`. Тем не менее, в C такая организация разрешена. Кроме того, как было описано в главе 15, язык C позволяет смешивать встраиваемые и внешние определения, что в C++ не допускается.

Средства C99/C11, которых нет в C++11

Хотя традиционно считается, что язык C в большей или меньшей степени является подмножеством C++, в стандарте C99 появились некоторые средства, отсутствующие в C++. Ниже перечислены наиболее заметные из них:

- назначенные инициализаторы;
- составные инициализаторы;
- ограниченные указатели;
- массивы переменной длины;
- гибкие элементы массивов;
- макросы с переменным количеством аргументов.

На заметку!

Приведенный список — это просто моментальный снимок на конкретный момент, и перечни разделяемых и неразделяемых возможностей продолжат видоизменяться. Например, в C++14 добавится средство, подобное массивам переменной длины C99.

B

Набор символов ASCII

Символы сохраняются в памяти компьютеров с использованием числовых кодов. В США наиболее часто применяется кодировка ASCII (American Standard Code for Information Interchange – американский стандартный код для обмена информацией). Язык С позволяет представить большинство одиночных символов напрямую путем заключения символа в одинарные кавычки, например, 'A' для символа *A*. Кроме того, одиночный символ можно представить с использованием его восьмеричного или шестнадцатеричного кода, перед которым должна находиться обратная косая черта, например, '\012' и '\0xa' соответствуют символу перевода строки (LF). Управляющие последовательности подобного рода также могут быть частью строки, скажем, такой: "Добро пожаловать,\012уважаемый".

В представленной ниже таблице символ ^, применяемый как префикс, обозначает клавишу <Ctrl>.

Десятичное представление	Восьмеричное представление	Шестнадцатеричное представление	Двоичное представление	Символ	Имя в коде ASCII
0	0	0	00000000	^@	NUL
1	01	0x1	00000001	^A	SOH
2	02	0x2	00000010	^B	STX
3	03	0x3	00000011	^C	ETX
4	04	0x4	00000100	^D	EOT
5	05	0x5	00000101	^E	ENQ
6	06	0x6	00000110	^F	ACK
7	07	0x7	00000111	^G	BEL
8	010	0x8	00001000	^H	BS
9	011	0x9	00001001	^I, табуляция	HT
10	012	0xa	00001010	^J	LF
11	013	0xb	00001011	^K	VT
12	014	0xc	00001100	^L	FF
13	015	0xd	00001101	^M	CR
14	016	0xe	00001110	^N	SO
15	017	0xf	00001111	^O	SI
16	020	0x10	00010000	^P	DLE
17	021	0x11	00010001	^Q	DC1
18	022	0x12	00010010	^R	DC2
19	023	0x13	00010011	^S	DC3
20	024	0x14	00010100	^T	DC4
21	025	0x15	00010101	^U	NAK
22	026	0x16	00010110	^V	SYN
23	027	0x17	00010111	^W	ETB

Десятичное представление	Восьмеричное представление	Шестнадцатеричное представление	Двоичное представление	Символ	Имя в коде ASCII
24	030	0x18	00011000	^X	CAN
25	031	0x19	00011001	^Y	EM
26	032	0x1a	00011010	^Z	SUB
27	033	0x1b	00011011	^[, esc	ESC
28	034	0x1c	00011100	^\	FS
29	035	0x1d	00011101	^]	GS
30	036	0x1e	00011110	^^	RS
31	037	0x1f	00011111	^_	US
32	040	0x20	00100000	пробел	SP
33	041	0x21	00100001	!	
34	042	0x22	00100010	"	
35	043	0x23	00100011	#	
36	044	0x24	00100100	$	
37	045	0x25	00100101	%	
38	046	0x26	00100110	&	
39	047	0x27	00100111	'	
40	050	0x28	00101000	(	
41	051	0x29	00101001	)	
42	052	0x2a	00101010	*	
43	053	0x2b	00101011	+	
44	054	0x2c	00101100	,	
45	055	0x2d	00101101	-	
46	056	0x2e	00101110		
47	057	0x2f	00101111	/	
48	060	0x30	00110000	0	
49	061	0x31	00110001	1	
50	062	0x32	00110010	2	
51	063	0x33	00110011	3	
52	064	0x34	00110100	4	
53	065	0x35	00110101	5	
54	066	0x36	00110110	6	
55	067	0x37	00110111	7	
56	070	0x38	00111000	8	
57	071	0x39	00111001	9	
58	072	0x3a	00111010	:	

Десятичное представление	Восьмеричное представление	Шестнадцатеричное представление	Двоичное представление	Символ	Имя в коде ASCII
59	073	0x3b	00111011	;	
60	074	0x3c	00111100	<	
61	075	0x3d	00111101	=	
62	076	0x3e	00111110	>	
63	077	0x3f	00111111	?	
64	0100	0x40	01000000	@	
65	0101	0x41	01000001	A	
66	0102	0x42	01000010	B	
67	0103	0x43	01000011	C	
68	0104	0x44	01000100	D	
69	0105	0x45	01000101	E	
70	0106	0x46	01000110	F	
71	0107	0x47	01000111	G	
72	0110	0x48	01001000	H	
73	0111	0x49	01001001	I	
74	0112	0x4a	01001010	J	
75	0113	0x4b	01001011	K	
76	0114	0x4c	01001100	L	
77	0115	0x4d	01001101	M	
78	0116	0x4e	01001110	N	
79	0117	0x4f	01001111	O	
80	0120	0x50	01010000	P	
81	0121	0x51	01010001	Q	
82	0122	0x52	01010010	R	
83	0123	0x53	01010011	S	
84	0124	0x54	01010100	T	
85	0125	0x55	01010101	U	
86	0126	0x56	01010110	V	
87	0127	0x57	01010111	W	
88	0130	0x58	01011000	X	
89	0131	0x59	01011001	Y	
90	0132	0x5a	01011010	Z	
91	0133	0x5b	01011011	[	
92	0134	0x5c	01011100	\	
93	0135	0x5d	01011101	]	

Десятичное представление	Восьмеричное представление	Шестнадцатеричное представление	Двоичное представление	Символ	Имя в коде ASCII
94	0136	0x5e	01011110	^	
95	0137	0x5f	01011111	_	
96	0140	0x60	01100000	`	
97	0141	0x61	01100001	a	
98	0142	0x62	01100010	b	
99	0143	0x63	01100011	c	
100	0144	0x64	01100100	d	
101	0145	0x65	01100101	e	
102	0146	0x66	01100110	f	
103	0147	0x67	01100111	g	
104	0150	0x68	01101000	h	
105	0151	0x69	01101001	i	
106	0152	0x6a	01101010	j	
107	0153	0x6b	01101011	k	
108	0154	0x6c	01101100	l	
109	0155	0x6d	01101101	m	
110	0156	0x6e	01101110	n	
111	0157	0x6f	01101111	o	
112	0160	0x70	01110000	p	
113	0161	0x71	01110001	q	
114	0162	0x72	01110010	r	
115	0163	0x73	01110011	s	
116	0164	0x74	01110100	t	
117	0165	0x75	01110101	u	
118	0166	0x76	01110110	v	
119	0167	0x77	01110111	w	
120	0170	0x78	01111000	x	
121	0171	0x79	01111001	y	
122	0172	0x7a	01111010	z	
123	0173	0x7b	01111011	{	
124	0174	0x7c	01111100	|	
125	0175	0x7d	01111101	}	
126	0176	0x7e	01111110	~	
127	0177	0x7f	01111111	del, стирание	

Предметный указатель

П

Р

С

Т

У



Ц

Ч

Ш

Я